

IN THE UNITED STATES PATENT AND TRADEMARK OFFICE

In re application of: Christopher F. Parker
Serial No.: 09/349,198
Filing Date: July 7, 1999
Art Unit: 3694
Confirmation No. 6293
Examiner: Ella Colbert
Title: *Database Table Recovery System*

Mail Stop Appeal Brief - Patents

Commissioner for Patents
P.O. Box 1450
Alexandria, VA 22313-1450

Dear Sir:

Appeal Brief

Appellant has appealed to the Board of Patent Appeals and Interferences (the "Board") from the decision of the Examiner mailed July 11, 2007, finally rejecting pending Claims 1-4 and 12-20. An Advisory Action was mailed November 28, 2007. Appellant filed a Notice of Appeal, with the statutory fee of \$510.00, on December 11, 2007. Appellant respectfully submits this Appeal Brief with the statutory fee of \$510.00.

Real Party in Interest

This Application is currently owned by Computer Associates Think, Inc. as indicated by an assignment recorded on April 3, 2000 in the Assignment Records of the United States Patent and Trademark Office (PTO) at Reel 010730, Frame 0981 (3 pages).

Related Appeals and Interferences

To the knowledge of Appellant's counsel, there are no appeals, interferences, or judicial proceedings that are related to or will directly affect, be directly affected by, or have a bearing on the Board's decision regarding this Appeal.

Status of Claims

Claims 1-6 and 12-20 are pending in this Application. Claims 1-4 and 12-20 stand rejected pursuant to a Final Office Action mailed July 11, 2007 (the "Final Office Action"). The Final Office Action indicates that independent Claims 5-6 are allowed. (Final Office Action at 6-7) Claims 1-4 and 12-20 are presented for appeal. All pending claims are shown in Appendix A, along with an indication of the status of those claims.

Status of Amendments

All amendments submitted by Appellant have been entered by the Examiner prior to the mailing of the Final Office Action.

Summary of Claimed Subject Matter

In certain embodiments, the present invention provides a database table recovery system. A database may use tables that depend from a tablespace. The tablespace contains all of the semi-permanent data of the database, and the tables contain user updates and modifications to that data. Users access the database data from the tablespace by creating a subset of that data in a table, and then modify and update the table data. The users periodically update the database to overwrite the data in the tablespace with the table data.

With conventional techniques, one problem that occurs when a user updates the data in a table is that the data updates may be incorrect, such as due to a loss of power during a table update, an error in a data input routine, or other common sources of error. In such situations, it is necessary to rebuild the data in the table without storing the table data to the tablespace, because the corrupted data must not be allowed to be transmitted to the tablespace. For example, it is common to keep a log record file of table updates, such that the tablespace data may be updated using the log record file.

One drawback with known methods of recovering tables is that they require that the tablespace be recovered with the log record files. When the tablespace is recovered, all access to the tablespace must be restricted, including access by unaffected tables. In addition, access to all tables must also be restricted until the tablespace is recovered. As a result, if two or more users are utilizing the tablespace to create two or more different tables, then all users are unable to access their tables while the data for a single table is recovered. This drawback may result in complete disruption of work while the tablespace is being recovered.

Fig. 1 illustrates an example tablespace with two dependent tables in accordance with one embodiment of the present invention. A tablespace 10 is broken down into four columns (i.e., Cols. A-D). In addition, tablespace 10 is broken down into five pages (i.e., Pages 1-5). The column definitions of each column are uniform across each row and page. The page size of each page is uniform, but may include varying numbers of rows depending upon the number of characters in variable column fields of each row. For example, if each page contains 4,096 bytes of data, then one page may contain two rows having variable fields that total 4,096 bytes, and another page may contain four rows having variable fields that total 4,096 bytes. Thus, example page one includes four rows, example page two includes two rows, example page three includes three rows, example page four includes four rows, and example page five includes two rows.

Example table 20 and example table 30 are drawn from tablespace 10. Table 20 includes column A data and column C data, while table 30 includes column B data and column D data. Table 20 and table 30 further include rows, but are not broken down into pages of data. In operation, table 20 may become corrupted due to improper data input, systems operation, or other error sources. Any updates that were made to table 20 since the time that table 20 was last read from tablespace 10 must therefore be incorporated into the data of tablespace 10 before table 20 can be reconstructed.

Tables 20 and 30 include log records 22 and 24, respectively. Each log record 22 and 24 is a record of changes that were made to its corresponding table. In addition, a backup copy of tablespace 10 is maintained by the database. The backup copy may be stored on tape or disk, or any other storage medium.

In accordance with one embodiment of the present invention, the updates made to table 20 are applied to the backup copy of tablespace 10 data, which allows table 30 to be accessed in a read-only mode while table 20 is being recovered. Thus, it is not necessary to apply the log records for table 20 and table 30 to tablespace 10 and thus modify tablespace 10 if the data for table 20 is corrupted but the data for table 30 is not corrupted. Instead, table 20 is rebuilt from the backup copy of tablespace 10 to which the updates from the table 20 log record file have been applied. In this manner, the configuration of table 20, table 30, and tablespace 10 are maintained in the state that they were in prior to the corruption of the table 20 data.

The present invention may eliminate the need to apply the log record files from table 20 and table 30 to tablespace 10 in order to recover table 20. Users may have read-only access to other tables such as table 30, which reduces the disruption of work that may result from recovery of the data in table 20. In particular, read-only access may be provided for table 30 to prevent any changes from being made to tablespace 10 while table 20 is being recovered. The backup copy of tablespace 10 is then stored into a working data memory. All of the rows of table 20 are then deleted, and any indices of table 20 are locked out from access to prevent changes to the indices from being made. The log records associated with table 20 are then read from the log.

After the log records are read, they are read to a log record workspace and sorted. The log records are then applied to the backup copy of tablespace 10. Sorting the log records decreases the number of input and output operations that must be made to the backup copy of

tablespace 10, which may decrease the amount of processing time for recovering table 20. After the log records have been applied to the backup copy of tablespace 10, new table data pages are built with the updated backup copy of tablespace 10. The new table data pages are then scanned for records, or database "rows," that belong to table 20. These records are selected and are used to update table 20. After table 20 is updated, update access to table 20, table 30 and tablespace 10 is allowed.

Fig. 2 illustrates an example table recovery system 200 according to one embodiment of the present invention. Table recovery system 200 may be used to recover a table in which corrupted data is stored without requiring tablespace 10 and all other tables of tablespace 10 to be recovered also. Table recovery system 200 thus returns tablespace 10 and tables to the configuration that they were in prior to the corruption of data in the affected table.

Table recovery system 200 may be implemented in hardware, software, or a suitable combination of hardware and software. Table recovery system 200 is coupled to tablespace backup storage 202, log record storage system 204, and table storage (spacemap) 206. Tablespace backup storage 202, log record storage system 204, and table storage 206 are disc storage mechanisms, random access memory, or other suitable data storage devices that are used to store tablespace backup data, log record data, and table data respectively.

Table recovery system 200 comprises individual subsystems that may each be implemented as software, hardware, or a suitable combination of software and hardware. Furthermore, the subsystems of table recovery system 200 may be individual functional aspects of a single system. For example, each of the subsystems of table recovery system 200 may be functions or commands in a DB2 database system, or may also or alternatively be logic circuits, programmable devices, or other suitable systems or components.

Log records sorter system 210 is used to sort log records obtained from log record storage system 204. Log records sorter system 210 optimizes input/output operations by grouping sets of log records according to data page and log records location. Data page updater system 212 is coupled to log records sorter system 210 and tablespace backup storage 202. Data page updater system 212 updates the backup copy of tablespace 10 from tablespace backup storage 202 with sorted log records received from log records sorter system 210.

Data page scanner system 214 is coupled to data page updater system 212 and is operable to scan the updated tablespace backup copy and identify table rows for extraction.

Page row extractor system 216 is coupled to data page scanner system 214, and extracts the page rows identified by the data page scanner system 214. Table row inserter system 218 receives the page rows from page row extractor system 216 and inserts them into the table that is being recovered. For example, the table that is being recovered may be stored on table storage system 206, which may be a space map that has been reinitiated with all rows deleted.

In operation, table recovery system 200 is used to recover a table from a tablespace without requiring all tables that depend from tablespace 10 to be rebuilt. For example, table recovery system 200 may operate in a DB2 Database System in which incorrect updates to a table have been made. If the table is not updated, then the incorrect updates will be imposed on tablespace 10, or the updates to the table will be lost. Table recovery system 200 is used to implement the updates to the table without requiring tablespace 10 and all other tables to be reconstructed. In this manner, the table data may be reconstructed while other tables can be accessed in a "read only" mode. Furthermore, each table and the table space is returned to the configuration that was present prior to the corruption of the data in the affected table.

Fig. 3 illustrates an example flow chart of steps executed by one embodiment of the present invention for recovering a table. At step 302, all updates to tablespace 10 from which the table depends are completed. For example, a DB2 "quiesce" function may be used to implement all queued updates to a tablespace, so as to prevent changes to tablespace 10 from occurring as table recovery is performed. At step 304, access to other tables that are dependent from tablespace 10 is set to "read only" mode. At step 306, the backup copy of tablespace 10 is received from backup storage to system storage. For example, the backup copy of tablespace 10 may be stored to a random access memory, a magnetic data storage medium, or other suitable data storage devices.

At step 308, all rows of the table that is to be reconstructed are deleted. At step 310, access to the indices of the table is locked out. The indices of the table may be used to verify that the data in the table has not changed after the recovery of the table has been completed. At step 312, log records are read from the log record storage associated with the table. The present invention implements those log updates that have occurred prior to table recovery on the copied tablespace, while leaving the original tablespace and all other dependent tables unaffected. In this manner, the table having corrupted data is returned to its original configuration, and no changes are implemented to tablespace 10 or other dependent tables.

At step 314, the log record file is copied to a log record work space. The log record work space is used at step 316 so that the log records may be sorted. At step 318, the log records are applied to tablespace 10 backup copy. At step 320, new table data pages are built with the updated tablespace backup copy. At step 322, the new table data pages are scanned for records that belong to the table that is being recovered. For example, each table comprises table keys that are columns from tablespace 10 that have been marked for use by the table. These table keys are extracted for each row. At step 324, the table is updated with the new table data pages from tablespace 10.

In operation, method 300 is used to recover a table after the table data has been corrupted without requiring tablespace 10 from which the table depended on to be recovered. Method 300 allows access to other tables of tablespace 10 without requiring those tables to be rebuilt. Method 300 optimizes the table recovery process by sorting the log records, such that input/output operations to tablespace 10 are optimized during the table recovery process.

The present invention allows recovery of a table without requiring the tablespace from which the table depends and all other tables that depend from the tablespace to be reconstructed. The present invention returns the reconstructed table to the configuration it was in prior to corruption of the data without also modifying the configuration of the tablespace and other dependent tables. In certain embodiments, the present invention provides a system and method for recovering a table that allow multiple users to access unaffected tables, and that do not require the tablespace to be recovered. Although the embodiments described allow users to access other tables in read-only mode, in other embodiments the users may be able to access the tables in update access mode while the table is recovered.

With regard to the independent claims currently under Appeal, Appellant provides the following concise explanation of the subject matter recited in the claim elements. For brevity, Appellant does not necessarily identify every portion of the Specification and drawings relevant to the recited claim elements. Additionally, this explanation should not be used to limit Appellant's claims but is intended to assist the Board in considering the Appeal of this Application.

For example, independent Claim 1 recites the following:

A system for recovering a database table (e.g., Fig. 2; Spec. at 3:9-4:3 and 7:21-10:20) comprising:

a database table recovery system (e.g., Fig. 2; Spec. at 7:21-10:20) operable to:

retrieve a backup copy of a tablespace (e.g., Spec. at 3:10-17, 5:19-6:2, 11:11-18);

apply updates to the backup copy from a log associated with a database table (e.g., Spec. at 3:10-17, 6:3-15, 7:7-13, 12:11-15); and

restore the database table associated with the tablespace from the updated backup copy without recovering the tablespace (e.g., Spec. at 3:10-4:3, 6:3-20, 7:7-20, 12:16-13:17); and

a tablespace access system coupled to the database table recovery system, wherein the tablespace access system is operable to restrict access to the tablespace to read-only access (e.g., Spec. at 6:3-7:6, 10:6-20, 11:11-18).

As another example, independent Claim 12 recites the following:

A method of recovering a first database table that depends on a tablespace (e.g., Fig. 3; Spec. at 3:9-4:3 and 10:21-13:17), said method comprising:

receiving a backup copy of a tablespace having data for one or more database tables (e.g., Spec. at 3:10-17, 5:19-6:2, 11:11-18);

reading log records associated with a first database table of the one or more database tables (e.g., Spec. at 3:10-17, 7:5-13, 11:19-12:10);

applying the log records to the backup copy without recovering the tablespace (e.g., Spec. at 3:10-17, 6:3-15, 7:7-13, 12:11-15);

building new table data pages from the backup copy (e.g., Spec. at 3:10-17, 7:14-20, 12:16-13:2);

scanning the new table data pages for records of the first database table (e.g., Spec. at 3:10-17, 7:14-20, 9:16-19, 12:16-13:2); and

updating the first database table from the records (e.g., Spec. at 3:10-17, 6:3-7:20, 10:1-20, 12:16-13:17).

As another example, independent Claim 17 recites the following:

A method for recovering a database table that depends on a tablespace (e.g., Fig. 3; Spec. at 3:9-4:3 and 10:21-13:17), said method comprising:

receiving a backup copy of a tablespace having data for one or more database tables (e.g., Spec. at 3:10-17, 5:19-6:2, 11:11-18);

reading log records associated with a first database table of the one or more database tables (e.g., Spec. at 3:10-17, 7:5-13, 11:19-12:10);

applying the log records to the backup copy without recovering the tablespace (e.g., Spec. at 3:10-17, 6:3-15, 7:7-13, 12:11-15);

building one or more table data pages from the backup copy having the log records applied (e.g., Spec. at 3:10-17, 7:14-20, 12:16-13:2);

selecting one or more records from the one or more database table data pages, the one or more records belonging to the first database table (e.g., Spec. at 7:14-20, 12:16-13:2); and

updating the first database table with the one or more records selected from the one or more table data pages, while allowing access to the rest of the one or more database tables in the tablespace (e.g., Spec. at 3:10-17, 6:3-7:20, 10:1-20, 12:16-13:17),

wherein the first database table can be recovered without having to recover the tablespace (e.g., Spec. at 3:18-4:3, 7:21-8:7, 10:6-15, 13:3-17).

Grounds of Rejection to be Reviewed on Appeal

Are Claims 1-4 and 12-20 patentable under 35 U.S.C. § 103(a) over U.S. Patent 5,721,915 to Sockut et al. ("*Sockut*") in view of U.S. Patent 5,517,641 to Barry, et al. ("*Barry*")?

Argument

For at least the following reasons, the Examiner's rejections of Claims 1-4 and 12-20 are improper and should be reversed by the Board. Appellant addresses each of independent Claims 1, 12, and 17.

I. Overview

Claims 1-4 and 12-20 stand rejected under 35 U.S.C. §103(a) as being unpatentable over *Sockut* in view of *Barry*, copies of which are attached as Appendices B and C, respectively. Appellant respectfully submits that these rejections are improper and should be reversed by the Board.

II. Legal Standard for Obviousness

The question raised under 35 U.S.C. § 103 is whether the prior art taken as a whole would suggest the claimed invention taken as a whole to one of ordinary skill in the art at the time of the invention. One of the three basic criteria that must be established by an Examiner to establish a *prima facie* case of obviousness is that "the prior art reference (or references when combined) must teach or suggest ***all the claim limitations***." See M.P.E.P. § 706.02(j) citing *In re Vaeck*, 947 F.2d 488, 20 U.S.P.Q.2d 1438 (Fed. Cir. 1991) (emphasis added). "***All words*** in a claim must be considered in judging the patentability of that claim against the prior art." See M.P.E.P. § 2143.03 citing *In re Wilson*, 424 F.2d 1382, 1385 165 U.S.P.Q. 494, 496 (C.C.P.A. 1970) (emphasis added).

In addition, even if all elements of a claim are disclosed in various prior art references, which is certainly not the case here as discussed below, the claimed invention taken as a whole still cannot be said to be obvious without some reason why one of ordinary skill at the time of the invention would have been prompted to modify the teachings of a reference or combine the teachings of multiple references to arrive at the claimed invention.

The controlling case law, rules, and guidelines repeatedly warn against using an applicant's disclosure as a blueprint to reconstruct the claimed invention. For example, the M.P.E.P. states, "The tendency to resort to 'hindsight' based upon applicant's disclosure is often difficult to avoid due to the very nature of the examination process. However,

impermissible hindsight must be avoided and the legal conclusion must be reached on the basis of the facts gleaned from the prior art.” M.P.E.P. ch. 2142.

The U.S. Supreme Court’s decision in *KSR Int’l Co. v. Teleflex, Inc.* reiterated the requirement that Examiners provide an explanation as to why the claimed invention would have been obvious. *KSR Int’l Co. v. Teleflex, Inc.*, 127 S.Ct. 1727 (2007). The analysis regarding an apparent reason to combine the known elements in the fashion claimed in the patent at issue “should be made explicit.” *KSR*, 127 S.Ct. at 1740-41. “Rejections on obviousness grounds cannot be sustained by mere conclusory statements; instead, there must be some articulated reasoning with some rational underpinning to support the legal conclusion of obviousness.” *Id.* at 1741 (internal quotations omitted).

The new examination guidelines issued by the PTO in response to the *KSR* decision further emphasize the importance of an explicit, articulated reason why the claimed invention is obvious. Those guidelines state, in part, that “[t]he key to supporting any rejection under 35 U.S.C. 103 is the clear articulation of the reason(s) why the claimed invention would have been obvious. The Supreme Court in *KSR* noted that the analysis supporting a rejection under 35 U.S.C. 103 should be made explicit.” *Examination Guidelines for Determining Obviousness Under 35 U.S.C. 103 in View of the Supreme Court Decision in KSR International Co. v. Teleflex Inc.*, 72 Fed. Reg. 57526, 57528-29 (Oct. 10, 2007) (internal citations omitted). The guidelines further describe a number of rationales that, in the PTO’s view, can support a finding of obviousness. *Id.* at 57529-34. The guidelines set forth a number of particular findings of fact that must be made and explained by the Examiner to support a finding of obviousness based on one of those rationales. *See id.*

III. Independent Claim 1 is Allowable over the *Sockut-Barry* Combination

A. The *Sockut-Barry* Combination Fails to Disclose, Teach, or Suggest Each and Every Limitation Recited in Claim 1

At a minimum, the proposed *Sockut-Barry* combination fails to disclose, teach, or suggest the database table recovery system recited in Claim 1 that is operable to:

- apply updates to the [retrieved] backup copy [of the tablespace] from a log associated with a database table; and

- restore the database table associated with the tablespace from the updated backup copy without recovering the tablespace.

As allegedly disclosing these limitations, the Examiner relies on various portions of *Sockut*. (Final Office Action at 3-4) Applicant respectfully disagrees that *Sockut* (whether considered alone or in combination with *Barry*) discloses, teaches, or suggests these limitations.

At the outset, Appellant notes that both *Sockut* and *Barry* appear to relate to *reorganizing* tablespaces, not *recovering* a database table. (See, e.g., *Sockut*, Abstract; Col. 1, ll. 11-15; Col. 3, l. 60 – Col. 4, l. 29; Col. 9, ll. 29-44; *Barry* at 2:6-19) Recovering a database table is the process of restoring a database table to a prior effective state, after the database has been erased or corrupted for example. (See, e.g., Specification at 2:1-10) Reorganizing a tablespace is the process of rearranging clusters to reduce the degree to which clusters are scattered. (See, e.g., *Barry* at 2:6-19)

Sockut discloses online reorganization of a database, and particularly the interaction between the application of a log and maintenance of a table that maps record identifiers (RIDs) during online reorganization of the database. (*Sockut*, 1:12-16) The Examiner cites Col. 9, ll. 19-22, which mentions the phrase “a backup copy of the new table space.” Appellant assumes the Examiner is equating this disclosure in *Sockut* with the backup tablespace recited in Claim 1. Appellant will assume for the sake of argument only (and not by way of concession or agreement) that the Examiner’s proposed equation is possible.

It appears to Appellant that the Examiner merely located the terms “backup” and “table space” in *Sockut*, but that the other disclosures in *Sockut* that the Examiner apparently equates with other limitations in Claim 1 (some of which reference *the backup copy of the tablespace*) do not relate to this disclosure of the purported backup copy in *Sockut*. For example, the cited disclosure of the backup copy of the new table space in *Sockut* appears in a description of a method. (See *Sockut*, 7:50-9:62) However, nowhere does this series of steps appear to disclose, teach, or suggest a database table recovery system that is operable to “apply updates to [a retrieved] backup copy [of the tablespace] from a log associated with a database table” and “restore the database table associated with the tablespace from the updated backup copy without recovering the tablespace,” as recited in Claim 1.

With respect to the recitation in Claim 1 of a database table recovery system that is operable to “restore the database table associated with the tablespace from the updated backup copy without recovering the tablespace,” it is not entirely clear which particular portion of *Sockut* the Examiner relies upon as allegedly disclosing this particular limitation of Claim 1. In any event, one cited portion of *Sockut* merely discloses *reorganization* of a database. (*Sockut*, 1:21-22) According to another cited portion relating to a *reorganization* strategy called fuzzy reorganization, a reorganizer records a current relative byte address of a log (which, according to *Sockut*, is a sequence of entries in a file (a region of storage) recording the changes that occur in a database). (See *Sockut*, 1:63-2:6; Final Office Action at 2-3) According to *Sockut*, the reorganization copies data from an old (original) area for the table space to a new area for the table space in reorganized form. Concurrently, users can use the DBMS’s normal facilities to read and write the old area, and the DBMS uses its normal facilities to record the writing in a log. (*Sockut*, 2:6-11; Final Office Action at 2-3) The reorganizer then reads the log and applies it to the new area to bring the new area up to date. (*Sockut*, 2:11-12)

However, nowhere do these cited portions disclose, teach, or suggest a database table recovery system that is operable to “restore the database table associated with the tablespace from the updated backup copy without recovering the tablespace,” as recited in Claim 1. In fact, it is not at all clear how reorganizing a tablespace into a new tablespace could equate to “**restor[ing]** the database table associated with the tablespace from the updated backup copy without **recovering** the tablespace,” as recited in Claim 1. *Sockut* discloses reorganizing a tablespace by copying a tablespace in use to a new area and applying logs to the newly copied tablespace. The cited portions describe manipulating record identifiers (RIDs) to be able to reorganize the tablespace in a new area. The cited portions of *Socket* do not disclose, teach, or suggest “**restor[ing]** the database table associated with the tablespace from the updated backup copy without **recovering** the tablespace,” as recited in Claim 1. Rather, the cited portions of *Socket* appear to disclose modifying the tablespace in order to reorganize it.

The Examiner also cites *Sockut* at 3:61-4:17. These cited portions appear to disclose how updates made during a reorganization of a database are applied to the reorganized

database. However, nowhere does this cited portion appear to disclose, teach, or suggest a database table recovery system that is operable to “restore the database table associated with the tablespace from the updated backup copy without recovering the tablespace,” as recited in Claim 1.

The Examiner also cited *Sockut* at 9:19-32, which recites, in part, “[a]t step 516, a backup copy of the new table space of partition (as a basis for further recovery) is created.” (Final Office Action at 9) Based on this teaching, the Examiner states, “Therefore, it is interpreted that Sockut does disclose the first two claim limitations of claim 1 and Barry discloses the other claim limitations of claim 1.” (Final Office Action at 9) Appellant assumes that by “the first two limitations of claim 1” the Examiner is referring to the recited database table recovery system that is operable to “retrieve a backup copy of a tablespace” and “apply updates to the backup copy from a log associated with a database table.” However, nowhere does the cited portion of *Sockut* disclose, teach, or suggest “apply[ing] updates to [a retrieved] backup copy [of the tablespace] from a log associated with a database table.”

Additionally, in the substantive rejection of Claim 1 (*see* Final Office Action at 2-3), the only limitation for which the Examiner relies on *Barry* is the limitation “a tablespace access system coupled to the table recovery system, the tablespace access system is operable to restrict access to the tablespace to read-only access,” as recited in Claim 1. The Examiner does not rely on *Barry* as disclosing “restor[ing] the database table associated with the tablespace from the updated backup copy without recovering the tablespace,” as the Examiner seems to imply in the Final Office Action “Response to Arguments” section. (Final Office Action at 9).¹ Appellant submits that neither the cited portion of *Sockut* nor *Barry* (which the Examiner has not cited for this limitation) discloses, teaches, or suggests “restor[ing] the database table associated with the tablespace from the updated backup copy

¹ The Examiner’s statement that “the Applicant’s are not reciting all of the claim limitations of claim 1 in the arguments” is not understood by Appellant. (Final Office Action at 9) Appellant is not obligated to recite each and every limitation recited in a claim when arguing the patentability of the claim. Respectfully, it is the Examiner’s burden to show that one or more references disclose, teach, or suggest each and every limitation in Appellant’s claim. “To establish *prima facie* obviousness of a claimed invention, **all the claim limitations** must be taught or suggested by the prior art.” M.P.E.P. ch. 2143.03 (emphasis added); *see also In re Royka*, 490 F.2d 981, 180 U.S.P.Q. 580 (C.C.P.A. 1974). Appellant is simply pointing out the perceived deficiencies in the Examiner’s rejection and is not required to reiterate each and every claim limitation to do so.

without recovering the tablespace,” as recited in Claim 1. Moreover, while the cited portion of *Sockut* (Col. 9, ll. 19-32) mentions creating a backup copy for “recoverability,” the balance of the cited portion appears to related to reorganizing a database (rather than recovering a database).

As another example, the Examiner acknowledges that *Sockut* fails to teach “a tablespace access system coupled to the table recovery system, the tablespace access system is operable to restrict access to the tablespace to read-only access,” as recited in Claim 1. (Final Office Action at 3) However, the Examiner argues that *Barry* teaches these limitations.

Whether or not the cited portions of *Barry* disclose, teach, or suggest “a tablespace access system coupled to the table recovery system, [wherein] the tablespace access system is operable to restrict access to the tablespace to read-only access,” as argued by the Examiner (and Appellant does not concede that *Barry* does disclose, teach, or suggest this element of Claim 1), Appellant demonstrates below that the Examiner has not provided an adequate reason either in the cited references or in the knowledge generally available to one of ordinary skill in the art at the time of Appellant’s invention to combine these references in the manner the Examiner proposes.

For at least these reasons, the proposed *Sockut-Barry* combination fails to disclose, teach, or suggest each and every limitation recited in independent Claim 1. Independent Claim 1 is allowable for at least this reason.

B. The *Sockut-Barry* Combination is Improper

Appellant maintains that the Examiner has not provided a sufficient basis, either in the cited references or in the knowledge generally available to one of ordinary skill in the art at the time of Appellant’s invention, to modify or combine *Sockut* with *Barry* in the manner the Examiner proposes. Appellant’s claims are allowable for at least this additional reason.

The Examiner states:

It would have been obvious to one having ordinary skill in the art at the time the invention was made to have a tablespace access system coupled to the table recovery system, [wherein] the tablespace access system is operable to restrict access to the tablespace to read-only access and in view of Sockut's teachings in col. 8, lines 57-67, col. 9, lines 1-18 and lines 37-44 of database performance and to modify in Sockut because such a modification would allow Sockut's system to have independent recovery of the data and indexes and a significant decrease in elapsed time since the log file updates are done for all objects in the database through the log file.

(Final Office Action at 3)

Appellant respectfully submits that the Examiner has not provided any support for the proposed basis that "such a modification would allow Sockut's system to have independent recovery of the data and indexes and a significant decrease in elapsed time since the log file updates are done for all objects in the database through the log file."

Moreover, it is entirely unclear and unexplained how the cited portions in *Sockut* even relate to the teachings that the Examiner is combining. For example, even assuming for the sake of argument only that *Barry* discloses "a tablespace access system coupled to the database table recovery system, wherein the tablespace access system is operable to restrict access to the tablespace to read-only access," as argued by the Examiner, it is entirely unclear why the alleged motivation of "allow[ing] Sockut's system to have independent recovery of the data and indexes and a significant decrease in elapsed time since the log file updates are done for all objects in the database through the log file" would lead one of ordinary skill in the art at the time of Appellant's invention to incorporate the teaching of "a tablespace access system coupled to the database table recovery system, wherein the tablespace access system is operable to restrict access to the tablespace to read-only access," as purportedly taught in *Barry*, into the system of *Sockut*. In other words, it is not clear how the alleged advantage of "allow[ing] Sockut's system to have independent recovery of the data and indexes and a significant decrease in elapsed time since the log file updates are done for all objects in the database through the log file" would even be achieved by modifying the system of *Sockut* to include "a tablespace access system coupled to the database table recovery system, wherein

the tablespace access system is operable to restrict access to the tablespace to read-only access,” as purportedly taught by *Barry*.

Instead, it appears that the Examiner has merely argued that one of ordinary skill in the art at the time the invention was made *could have* modified *Sockut* to perform the acknowledged deficient limitations (a point which Appellant does not concede). However, it does not appear to Appellant that the Examiner has pointed to any portions of the cited references that would explain why one of ordinary skill in the art at the time of invention would incorporate “a tablespace access system coupled to the table recovery system, [] the tablespace access system [being] operable to restrict access to the tablespace to read-only access,” as recited in Claim 1, into the particular techniques disclosed in *Sockut* (***without using Appellant’s claims as a guide for doing so***). See M.P.E.P. ch. 2143.01(III) (stating that the mere fact that references can be combined or modified does not render the resultant combination obvious unless the prior art also suggests the desirability of the combination); see also *In re Mills*, 916 F.2d 680, 16 U.S.P.Q.2d 1430 (Fed. Cir. 1990). Most recently, this requirement has been reaffirmed in an official USPTO memorandum dated May 3, 2007 wherein the Deputy Commissioner for Patent Operations pointed to sections of *KSR v. Teleflex*, which recite, “it will be necessary . . . to determine whether there was an ***apparent reason*** to combine the known elements in the fashion claimed by the patent at issue.”² Appellant submits that the statements made by the Examiner does not provide a supported explanation as to: (1) why it would have been obvious to one of ordinary skill in the art at the time of Appellant’s invention (***without using Appellant’s claims as a guide***) to modify *Sockut* in the manner proposed by the Examiner; and (2) how one of ordinary skill in the art at the time of Appellant’s invention would have actually done so.

The Examiner states that the argument “is not persuasive because a suggestion/motivation need not be expressly stated in one or all of the references used to show obviousness.” (Final Office Action at 7) This is a position Appellant has never argued. Instead, Appellant stated that the Examiner had “not provided a sufficient teaching, suggestion, or motivation, ***either in the cited references or in the knowledge generally available to one of ordinary skill in the art at the time of Applicant’s invention*** to modify or

² *KSR Int’l. Co v. Teleflex Inc.*, 550 U.S. ___, 82 U.S.P.Q.2d 1384 (2007) (emphasis added).

combine *Sockut* with *Barry* in the manner the Examiner proposes.” (See Response mailed March 21, 2007, at 10)

Appellant respectfully submits that the Examiner’s attempt to combine *Sockut* with *Barry* appears to constitute the type of impermissible hindsight reconstruction of Appellant’s claims, using Appellant’s claims as a blueprint, that is specifically prohibited by the M.P.E.P. and governing Federal Circuit cases.

Accordingly, since the Examiner has not demonstrated an adequate reason to combine *Sockut* and *Barry* in the manner the Examiner proposes, Appellant respectfully submits that the Examiner’s conclusions set forth in the Office Action do not meet the requirements set forth in the M.P.E.P. and the governing Federal Circuit case law for demonstrating a *prima facie* case of obviousness. Appellant respectfully submits that the rejection must therefore be withdrawn.

For at least these reasons, Appellant respectfully submits that the proposed *Sockut-Barry* combination is improper. Independent Claim 1 and its dependent claims are allowable for at least this additional reason.

C. Conclusions with Respect to Claim 1

For at least these reasons, Appellant respectfully submits that the Examiner has not established a *prima facie* case of obviousness based on the proposed *Sockut-Barry* combination with respect to independent Claim 1. Thus, for at least these reasons, Appellant submits that these rejections are improper and respectfully request that the Board reverse these rejections of independent Claim 1 and its dependent claims.

II. Independent Claim 12 is Allowable over the *Sockut-Barry* Combination

A. The *Sockut-Barry* Combination Fails to Disclose, Teach, or Suggest Each and Every Limitation Recited in Claim 12

Appellant reiterates that both *Sockut* and *Barry* appear to relate to *reorganizing* tablespaces, not *recovering* a database table. Recovering a database table is the process of restoring a database table to a prior effective state, after the database has been erased or

corrupted for example. (*See, e.g.*, Specification at 2:1-10) Reorganizing a tablespace is the process of rearranging clusters to reduce the degree to which clusters are scattered. (*See, e.g.*, *Barry* at 2:6-19)

The proposed *Sockut-Barry* combination fails to disclose, teach, or suggest at least the following limitations recited in Claim 17:

- reading log records associated with a first database table of the one or more database tables;
- applying the log records to the backup copy without recovering the tablespace;
- building new table data pages from the backup copy;
- scanning the new table data pages for records of the first database table; and
- updating the first database table from the records.

For example, as allegedly disclosing “reading log records associated with a first database table of the one or more database tables,” the Examiner cites *Sockut* at 4:5-11. The cited portion of *Sockut* states the following:

The reorganization in accordance with the present invention reads the log (that has been written to during reorganization) and applies the log to the new area to bring the new area up to date. However, after reading the log but before applying the logged writing, the reorganization of the present invention sorts the log entries by record identifier (RID).

(*Sockut* at 4:5-11) However, the cited portion fails to disclose, teach, or suggest “reading log records *associated with a first database table of the one or more database tables*,” as specifically recited in Claim 12.

Oddly, it appears that the Examiner also acknowledges that *Sockut* does not teach this entire element of Claim 12 (“reading log records associated with a first database table of the one or more database tables [of the tablespace]”), but the Examiner concludes that it would have been obvious to modify *Sockut* to include this element of Claim 12. (*See* Final Office Action at 3-4) Appellant notes that in the rejection of Claim 12, the Examiner does not reference *Barry* as purportedly making up for this deficiency of *Sockut*, nor does the Examiner cite *Barry* as providing a purported reason for modifying *Sockut* in the manner

proposed by the Examiner. (Final Office Action at 3-4)³ For at least those reasons discussed below, Appellant respectfully submits that the Examiner's proposed modifications to *Sockut* are improper.

As another example, the proposed *Sockut-Barry* combination fails to disclose, teach, or suggest "applying the log records to the backup copy without recovering the tablespace," as recited in Claim 12. At least because the proposed *Sockut-Barry* combination fails to disclose, teach, or suggest "reading *log records associated with a first database table* of the one or more database tables," as recited in Claim 12 and as acknowledged by the Examiner, the proposed *Sockut-Barry* combination necessarily fails to disclose, teach, or suggest "applying *the log records* to the backup copy without recovering the tablespace," as recited in Claim 12.

The Examiner apparently cites *Sockut* at 4:22-29 and 9:19-32 as allegedly disclosing these limitations. The first of these cited portions discloses the following:

To apply a log entry to the new area, the record in the new area to which the entry should apply must be identified (i.e., the new RID). In the present invention, identification of the new record (by new RID) is done by maintaining a temporary table that maps between the old and new RIDs. The strategy uses this table to translate log entries before sorting them (by new RID) and applying them to the new area.

(*Sockut* at 4:22-29) While this portion of *Sockut* discloses "apply[ing] a log entry to the new area," the cited portion does not disclose, teach, or suggest "applying the log records *to the backup copy [of the tablespace] without recovering the tablespace*," as specifically recited in Claim 12.

The second of these cited portions discloses the following:

At step 516, a backup copy of the new table space or partition (as a basis for future recoverability) is created. Read/write access to the new area of the table space or partition is started. Possible sequences for this step 614 include: (1) start read-only access, create a backup copy while allowing read-only access, and then start read/write access (after the backup copying completes); or (2) start the creation of a backup copy via a facility that allows concurrent

³ In fact, it is not entirely clear whether the rejection of Claim 12 (and various other claims, such as independent Claim 17) is based at all on *Barry* or is an obviousness rejection based solely on *Sockut*.

writing, and start read/write access as soon as the backup copying begins (instead of waiting for the backup copying to complete); or (3) create a backup copy during reorganization step 504, append translated log entries to the original log in steps 506 and 510, and start read/write access immediately in step 516.

(*Sockut* at 9:19-32) While this cited portion mentions creating a backup copy of a new tablespace (i.e., of the new, reorganized database), the cited portion makes no mention of applying any log records to the backup copy of the tablespace without recovering the tablespace, as recited in Claim 12.

As another example, the proposed *Sockut-Barry* combination fails to disclose, teach, or suggest “building new table data pages from the backup copy,” as recited in Claim 12. As allegedly disclosing these limitations, the Examiner cites the following statement from *Sockut*: If the log entry is an update from pointer to regular data record then proceed as follows. (Final Office Action at 4, citing *Sockut* 14:66-67) However, nowhere does this statement disclose, teach, or suggest “building new table data pages from the backup copy,” as recited in Claim 12. Indeed the statement does not even mention building anything, let alone building new table data pages from the backup copy (of the tablespace), as recited in Claim 12.

As another example, the proposed *Sockut-Barry* combination fails to disclose, teach, or suggest “scanning the new table data pages for records of the first database table,” as recited in Claim 12. As allegedly disclosing these limitations, the Examiner cites Col. 11, ll. 52-66 of *Sockut*. (Final Office Action at 4) While the cited portion of *Sockut* mentions the words “scan” or “scanning,” nowhere does the cited portion appear to disclose, teach, or suggest “scanning *the new table data pages for records of the first database table*,” as recited in Claim 12. In particular, Appellant respectfully asks the Examiner: Where in this cited portion does *Sockut* disclose the new table data pages and that such new table data pages are scanned (and particularly that they are scanned for records of the first database table)?

In an Advisory Action mailed November 28, 2007, the Examiner cites *Sockut* at 14:58-15:12 and 16:51-18:55 as allegedly disclosing these limitations. The first of these cited

portions relates to translation of an update entry found in the log. The cited portion discusses how to process certain types of log entries. However, it does not appear to Appellant that the cited portion discloses, teaches, or suggests “scanning *the new table data pages [built from the backup copy of the tablespace to which log records associated with the first database table have been applied without recovering the tablespace]* for records of the first database table,” as recited in Claim 12.

The second cited portion discusses that a *reorganizer* scans the table space and then sorts by a clustering key. This lengthy cited portion then appears to walk through example scenarios of the *reorganization* process. However, it does not appear to Appellant that the cited portion discloses, teaches, or suggests “scanning *the new table data pages [built from the backup copy of the tablespace to which log records associated with the first database table have been applied without recovering the tablespace]* for records of the first database table,” as recited in Claim 12.

As another example, the proposed *Sockut-Barry* combination fails to disclose, teach, or suggest “updating the first database table from the records,” as recited in Claim 12. At least because the proposed *Sockut-Barry* combination fails to disclose, teach, or suggest “scanning the new table data pages for records of the first database table,” as recited in Claim 12, the proposed *Sockut-Barry* combination necessarily fails to disclose, teach, or suggest “updating the first database table from *the records [records of the first database table for which the new table data pages were scanned]*,” as recited in Claim 12.

For at least these reasons, Appellant respectfully submits that the proposed *Sockut-Barry* combination fails to disclose, teach, or suggest each and every limitation recited in independent Claim 12. Independent Claim 12 and its dependent claims are allowable for at least this reason.

B. The Proposed Modifications to *Sockut* are Improper

Appellant respectfully submits that the Examiner has not demonstrated an adequate reason, either in the cited references or in the knowledge generally available to one of ordinary skill in the art at the time of the invention for modifying *Sockut* in the manner

proposed by the Examiner. Appellant reiterates the above-discussed heavy burden incumbent on the Examiner for demonstrating a *prima facie* case of obviousness.

With respect to the rejection of Claim 12 and the proposed modification of *Sockut*, the Examiner states:

Sockut failed to teach reading log records associated with a first database table in the one or more database tables, but it would have been obvious to one having ordinary skill in the art at the time the invention was made to read log records associated with a first table in the one or more tables and to modify in Sockut because such a modification would allow the data to be read and updated in the first table before it is copied to the new table/tables and a backup copy is made of the data pages.

(Final Office Action at 4)

Appellant respectfully submits that the Examiner has not provided any support for this proposed basis for modifying *Sockut*. It appears that the Examiner has merely argued that one of ordinary skill in the art at the time the invention was made *could have* modified *Sockut* to perform the acknowledged deficient limitations (a point which Appellant does not concede). However, it does not appear to Appellant that the Examiner has pointed to any portions of the cited references that would teach, suggest, or motivate one of ordinary skill in the art at the time of invention to actually incorporate "reading log records associated with a first database table in the one or more database tables," as recited in Claim 12, into the particular techniques disclosed in *Sockut* (***without using Appellant's claims as a guide for doing so***). See M.P.E.P. ch. 2143.01(III) (stating that the mere fact that references can be combined or modified does not render the resultant combination obvious unless the prior art also suggests the desirability of the combination); *see also In re Mills*, 916 F.2d 680, 16 U.S.P.Q.2d 1430 (Fed. Cir. 1990). In other words, the statements made by the Examiner does not provide a supported explanation as to: (1) why it would have been obvious to one of ordinary skill in the art at the time of Appellant's invention to modify *Sockut* to incorporate "reading log records associated with a first database table in the one or more database tables," as recited in Claim 1; and (2) how one of ordinary skill in the art at the time of Appellant's invention would have actually done so.

There is certainly no reason to assume that one of ordinary skill in the art at the time of Appellant's invention would have been motivated to make the proposed modifications to *Sockut*. Therefore, it certainly would not have been obvious to one of ordinary skill in the art at the time of invention *to even attempt* to, let alone *to actually*, modify *Sockut* in the manner proposed by the Examiner.⁴ Appellant respectfully submits that the Examiner's attempt to modify *Sockut* appears to constitute the type of impermissible hindsight reconstruction of Appellant's claims, using Appellant's claims as a blueprint, that is specifically prohibited by the M.P.E.P. and governing Federal Circuit cases.

Accordingly, since the Examiner has not demonstrated an adequate reason to modify or combine *Sockut* and *Barry* in the manner the Examiner proposes, Appellant respectfully submits that the Examiner's conclusions set forth in the Office Action do not meet the requirements set forth in the M.P.E.P. and the governing Federal Circuit case law for demonstrating a *prima facie* case of obviousness. Appellant respectfully submits that the rejection must therefore be withdrawn.

For at least these reasons, Appellant respectfully submits that the proposed *Sockut-Barry* combination is improper. Independent Claim 12 and its dependent claims are allowable for at least this additional reason.

C. Conclusions with Respect to Claim 12

For at least these reasons, Appellant respectfully submits that the Examiner has not established a *prima facie* case of obviousness based on the proposed *Sockut-Barry* combination with respect to independent Claim 12. Thus, for at least these reasons, Appellant submits that these rejections are improper and respectfully request that the Board reverse these rejections of independent Claim 12 and its dependent claims.

⁴ If "common knowledge" or "well known" art is relied upon by the Examiner to combine or modify the references, Appellant respectfully requests that the Examiner provide a reference pursuant to M.P.E.P. § 2144.03 to support such an argument. If the Examiner relies on personal knowledge to supply the required motivation or suggestion to combine or modify the references, Appellant respectfully requests that the Examiner provide an affidavit supporting such facts pursuant to M.P.E.P. § 2144.03.

III. Independent Claim 17 is Allowable over the *Sockut-Barry* Combination

A. The *Sockut-Barry* Combination Fails to Disclose, Teach, or Suggest Each and Every Limitation Recited in Independent Claim 17

Appellant notes that in rejecting Claim 17, the Examiner references the rejection of Claim 12. (Final Office Action at 6) Thus, Appellant cites and references the Examiner's rejection of Claim 12.

Appellant reiterates that both *Sockut* and *Barry* appear to relate to *reorganizing* tablespaces, not *recovering* a database table. Recovering a database table is the process of restoring a database table to a prior effective state, after the database has been erased or corrupted for example. (*See, e.g.*, Specification at 2:1-10) Reorganizing a tablespace is the process of rearranging clusters to reduce the degree to which clusters are scattered. (*See, e.g.*, *Barry* at 2:6-19)

The proposed *Sockut-Barry* combination fails to disclose, teach, or suggest at least the following limitations recited in Claim 17:

- reading log records associated with a first database table of the one or more database tables;
- applying the log records to the backup copy without recovering the tablespace;
- building one or more table data pages from the backup copy having the log records applied;
- selecting one or more records from the one or more database table data pages, the one or more records belonging to the first database table; and
- updating the first database table with the one or more records selected from the one or more table data pages, while allowing access to the rest of the one or more database tables in the tablespace,
- wherein the first database table can be recovered without having to recover the tablespace.

For example, for at least certain reasons analogous to those discussed above with reference to independent Claim 12, the proposed *Sockut-Barry* combination fails to disclose, teach, or suggest “reading log records associated with a first database table of the one or more database tables,” “applying the log records to the backup copy without recovering the tablespace,” “building one or more table data pages from the backup copy having the log records applied,” “selecting one or more records from the one or more database table data

pages, the one or more records belonging to the first database table,” and “updating the first database table with the one or more records selected from the one or more table data pages, while allowing access to the rest of the one or more database tables in the tablespace,” as recited in Claim 17.

As another example, the proposed *Sockut-Barry* combination fails to disclose, teach, or suggest “wherein the first database table can be recovered without having to recover the tablespace,” as recited in Claim 17. Given that the Examiner simply referenced the rejection of Claim 12 when rejecting Claim 17, it is not entirely clear what particular portions of *Sockut* or *Barry* the Examiner believes allegedly disclose this limitation of Claim 17. However, Appellant respectfully submits that none of the cited portions appear to disclose, teach, or suggest this limitation of Claim 17. As discussed above, *Sockut* discloses reorganizing a tablespace by copying a tablespace in use to a new area and applying logs to the newly copied tablespace. The cited portions describe manipulating record identifiers (RIDs) to be able to reorganize the tablespace in a new area. The cited portions of *Socket* do not disclose, teach, or suggest “wherein the first database table can be recovered without having to recover the tablespace,” as recited in Claim 17. Rather, the cited portions of *Socket* appear to disclose modifying the tablespace in order to reorganize it.

The Examiner acknowledges that *Sockut* does not teach an entire element of Claim 17, but the Examiner concludes that it would have been obvious to modify *Sockut* to include this element of Claim 12. (Final Office Action at 4) Appellant notes that in the rejection of Claim 12 (and thus Claim 17), the Examiner does not reference *Barry* as purportedly making up for this deficiency of *Sockut*, nor does the Examiner cite *Barry* as providing a purported reason for modifying *Sockut* in the manner proposed by the Examiner. (Final Office Action at 4) For at least those reasons discussed below, Appellant respectfully submits that the Examiner’s proposed modifications to *Sockut* are improper.

For at least these reasons, Appellant respectfully submits that the proposed *Sockut-Barry* combination fails to disclose, teach, or suggest each and every limitation recited in independent Claim 17. Independent Claim 17 and its dependent claims are allowable for at least this reason.

B. The Proposed Modifications to *Sockut* are Improper

For at least those reasons discussed above with reference to Claim 12, Appellant respectfully submits that the Examiner has not demonstrated an adequate reason, either in the cited references or in the knowledge generally available to one of ordinary skill in the art at the time of the invention for modifying *Sockut* in the manner proposed by the Examiner. Appellant reiterates the above-discussed heavy burden incumbent on the Examiner for demonstrating a *prima facie* case of obviousness. For at least these reasons, Appellant respectfully submits that the proposed modifications to *Sockut* are improper. Independent Claim 17 and its dependent claims are allowable for at least this additional reason.

C. Conclusions with Respect to Claim 17

For at least these reasons, Appellant respectfully submits that the Examiner has not established a *prima facie* case of obviousness based on the proposed *Sockut-Barry* combination with respect to independent Claim 17. Thus, for at least these reasons, Appellant submits that these rejections are improper and respectfully request that the Board reverse these rejections of independent Claim 17 and its dependent claims.

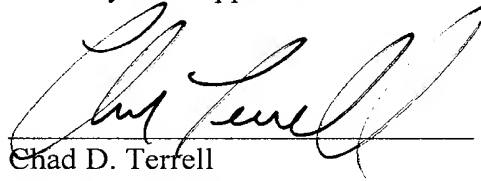
Conclusion

Appellant has demonstrated that, for at least the foregoing reasons, the present invention, as claimed, is clearly patentable over the references cited by the Examiner. Therefore, Appellant respectfully requests the Board to reverse the final rejection of the Examiner and instruct the Examiner to issue a Notice of Allowance of all pending claims.

The Commissioner is hereby authorized to charge the large entity fee of \$510.00 under 37 C.F.R. §§1.191(a) and 1.17(b) for filing this Appeal Brief to Deposit Account No. 02-0384 of Baker Botts L.L.P. Although no other fees are believed to be due at this time, the Commissioner is hereby authorized to charge any necessary additional fees and/or credit any overpayments to Deposit Account No. 02-0384 of Baker Botts L.L.P.

Respectfully submitted,

BAKER BOTTS L.L.P.
Attorneys for Appellant

A handwritten signature in black ink, appearing to read "Chad D. Terrell", is written over a horizontal line.

Chad D. Terrell
Reg. No. 52,279

Date: February 11, 2008

Customer Number: **05073**

Appendix A: The Claims

1. (Rejected) A system for recovering a database table comprising:
a database table recovery system operable to:
retrieve a backup copy of a tablespace;
apply updates to the backup copy from a log associated with a database table;
and
restore the database table associated with the tablespace from the updated
backup copy without recovering the tablespace; and
a tablespace access system coupled to the database table recovery system, wherein the
tablespace access system is operable to restrict access to the tablespace to read-only access.
2. (Rejected) The system of claim 1 wherein the database table recovery system
further comprises a log record sorter system operable to sort log records from the log.
3. (Rejected) The system of claim 2 further comprising:
a data page updater system coupled to the log record sorter system operable to apply
the log record updates to the backup copy.
4. (Rejected) The system of claim 3 further comprising:
a data page scanner system coupled to the data page updater system, the data page
scanner system operable to locate page rows associated with the database table in at least one
data page.

5. (Allowed) A system for recovering a database table comprising:

a database table recovery system, the database table recovery system operable to retrieve a backup copy of a tablespace and to apply updates to the backup copy from a log associated with a database table, and to restore the database table associated with the tablespace from the updated backup copy without recovering the tablespace, the database table recovery system comprising a log record sorter system operable to sort log records from the log;

a tablespace access system coupled to the database table recovery system, wherein the tablespace access system is operable to restrict access to the tablespace to read-only access;

a data page updater system coupled to the log record sorter system and operable to apply the log record updates to the backup copy;

a data page scanner system coupled to the data page updater system and operable to locate page rows associated with the database table in at least one data page; and

a page row extractor system coupled to the data page scanner system and operable to extract the page rows from the at least one data page that has been located by the data page scanner system.

6. (Allowed) A system for recovering a database table comprising:

a database table recovery system, the database table recovery system operable to retrieve a backup copy of a tablespace and to apply updates to the backup copy from a log associated with a database table, and to restore the database table associated with the tablespace from the updated backup copy without recovering the tablespace, the database table recovery system comprising a log record sorter system operable to sort log records from the log;

a tablespace access system coupled to the database table recovery system, wherein the tablespace access system is operable to restrict access to the tablespace to read-only access;

a data page updater system coupled to the log record sorter system and operable to apply the log record updates to the backup copy;

a data page scanner system coupled to the data page updater system and operable to locate page rows associated with the database table in at least one data page;

a page row extractor system coupled to the data page scanner system and operable to extract the page rows from the at least one data page that has been located by the data page scanner system; and

a table row inserter system coupled to the page row extractor system and operable to write the extracted page rows to the database table.

7-11. (Canceled).

12. (Rejected) A method of recovering a first database table that depends on a tablespace, said method comprising:

receiving a backup copy of a tablespace having data for one or more database tables;
reading log records associated with a first database table of the one or more database tables;

applying the log records to the backup copy without recovering the tablespace;

building new table data pages from the backup copy;

scanning the new table data pages for records of the first database table; and

updating the first database table from the records.

13. (Rejected) The method of claim 12, further comprising:
limiting access of a second table to the tablespace to read-only before the first table is updated, wherein the second table depends on the tablespace.
14. (Rejected) The method of claim 13, further comprising:
providing update access to the second table after the first table is updated.
15. (Rejected) The method of claim 12, further comprising sorting the log records.
16. (Rejected) The method of claim 12 wherein the first table, comprises rows and indices, the method further comprising:
deleting the rows; and
locking out the indices.
17. (Rejected) A method for recovering a database table that depends on a tablespace, said method comprising:
receiving a backup copy of a tablespace having data for one or more database tables;
reading log records associated with a first database table of the one or more database tables;
applying the log records to the backup copy without recovering the tablespace;
building one or more table data pages from the backup copy having the log records applied;
selecting one or more records from the one or more database table data pages, the one or more records belonging to the first database table; and
updating the first database table with the one or more records selected from the one or more table data pages, while allowing access to the rest of the one or more database tables in the tablespace,
wherein the first database table can be recovered without having to recover the tablespace.

18. (Rejected) The method of claim 17, further comprising:
allowing said at least one of the one or more database tables to have update access to the tablespace when the first database table is restored.

19. (Rejected) The method of claim 17, further comprising:
deleting all of the rows of the first database table; and
locking out access to indices of the first database table.

20. (Rejected) The method of claim 17, wherein applying log records to the tablespace backup copy further comprises:
reading the log records from a log record file to a log record workspace;
sorting the log records; and
applying the log records to the tablespace backup copy.

ATTORNEY DOCKET NO.
063170.6564

PATENT APPLICATION
USSN 09/349,198

Appendix B: *Sockut*



US005721915A

United States Patent [19]

Sockut et al.

[11] **Patent Number:** 5,721,915[45] **Date of Patent:** Feb. 24, 1998

[54] **INTERACTION BETWEEN APPLICATION OF A LOG AND MAINTENANCE OF A TABLE THAT MAPS RECORD IDENTIFIERS DURING ONLINE REORGANIZATION OF A DATABASE**

[75] **Inventors:** Gary Howard Sockut, San Jose;
Thomas Abel Beavin, Milpitas, both of Calif.

[73] **Assignee:** International Business Machines Corporation, Armonk, N.Y.

[21] **Appl. No.:** 457,150

[22] **Filed:** Jun. 1, 1995

Related U.S. Application Data

[63] Continuation of Ser. No. 366,564, Dec. 30, 1994.

[51] **Int. Cl.⁶** G06F 17/30

[52] **U.S. Cl.** 395/616; 395/617; 395/618;
395/202

[58] **Field of Search** 364/413.01; 395/616;
395/617, 618, 202, 600

[56] **References Cited**

U.S. PATENT DOCUMENTS

5,307,262 4/1994 Ertel 364/413.01

OTHER PUBLICATIONS

R.A. Crus, "Data Recovery in IBM Database 2", IBM Systems Journal, vol. 23, No. 2, 1984.

C. Mohan, "Algorithms for Creating Indexes for Very Large Tables Without Quiescing Updates", ACM Sigmod, 1992.

Scheuermann, "Concurrent File Reorganization for Record Clustering: A Performance Study", IEEE 1992.

Omiecinski et al., "Performance Analysis of a Concurrent File Reorganization Algorithm for Record Clustering", IEEE Transactions vol. 6, No. 2 Apr. 1994.

Wiener et al., "Bulk Loading into an OODB: A Performance Study," Proceedings of the 20th VLDB Conference, Santiago, Chile, 1994, pp. 120-131.

Performance Analysis of a Concurrent File Reorganization Algorithm for Record Clustering, *IEEE Transactions on Knowledge and Data Engineering*, vol. 6, No. 2, pp. 248-257, Apr. 1994.

Concurrent File Reorganization for Record Clustering: A Performance Study, *IEEE*, pp. 265-272, Jul. 1992.

Data Recovery in IBM Database 2, *IBM Systems Journal*, vol. 23, No. 2, pp. 178-188, 1984.

C. Mohan and Inderpal Narang, *Algorithms for Creating Indexes for Very Large Tables Without Quiescing Updates*, Database Technology Institute, pp. 361-370, Jun. 1992.

Primary Examiner—Thomas G. Black

Assistant Examiner—Cheryl R. Lewis

Attorney, Agent, or Firm—Sterne, Kessler, Goldstein & Fox P.L.L.C.; Marilyn Smith Dawkins, Esq.

[57] **ABSTRACT**

The present invention includes reorganization of a Database Management System (DBMS). The reorganization of the present invention is implemented by recording a first current Relative Byte Address (RBA). Then, data is copied from the old area in the table space to a new area in the table space in reorganized form. In the present invention, throughout most of reorganization a user maintains access to the DBMS's normal facilities to read and write to the old area. The DBMS uses its normal facilities to record writing, which occurs during reorganization, in a log. The reorganization in accordance with the present invention reads the log (that has been written to during reorganization) and processes the log to the new area to bring the new area up to date. This process is performed with the use of a RID mapping table. Finally, at the end of reorganization, the user's access is switched from the old area to the new area.

22 Claims, 6 Drawing Sheets

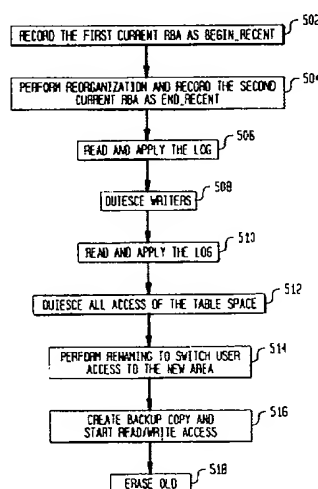
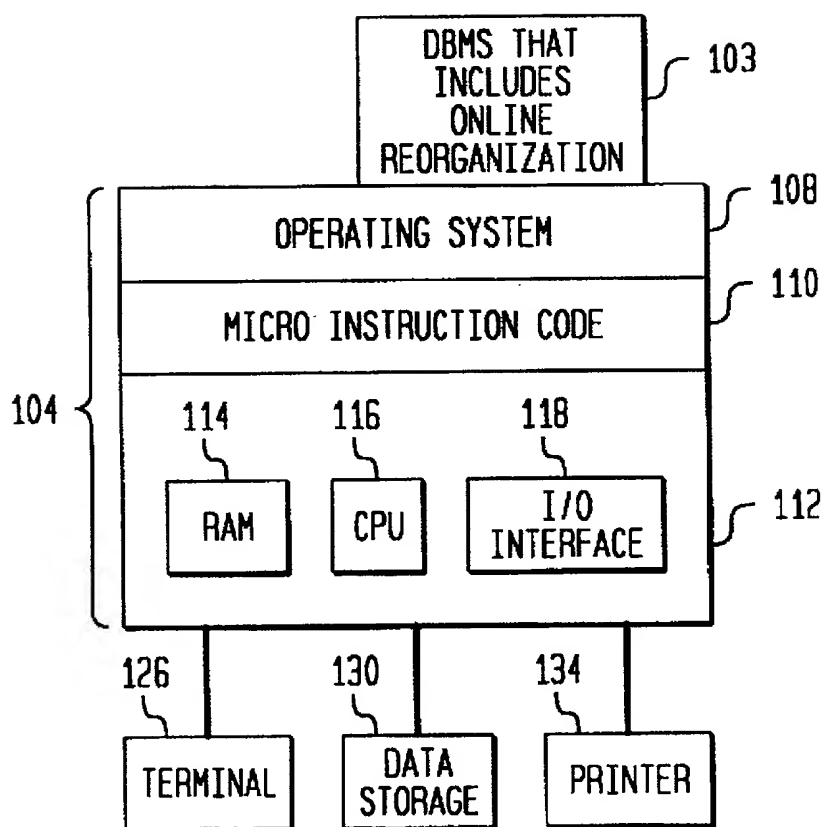


FIG. 1

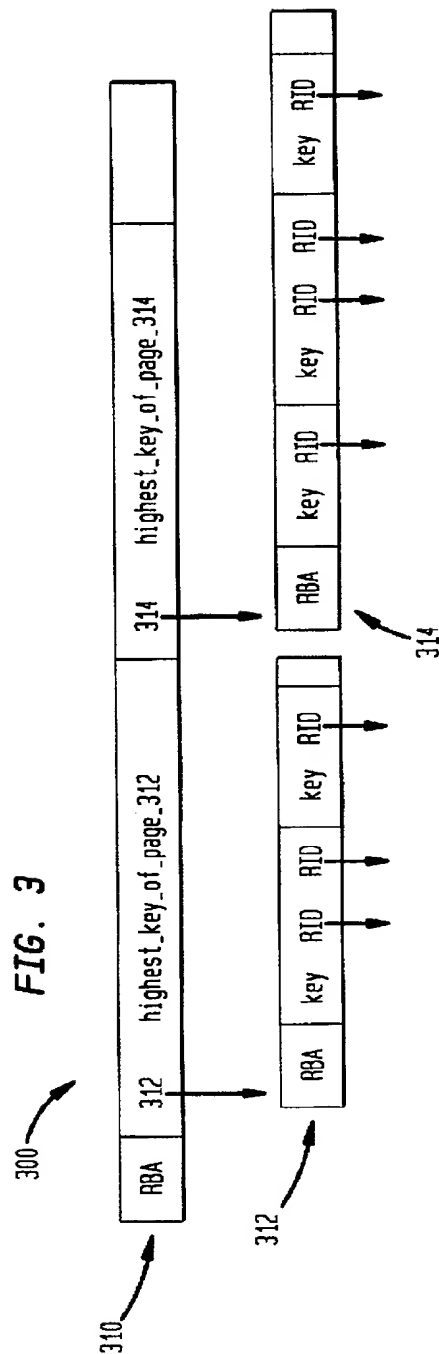
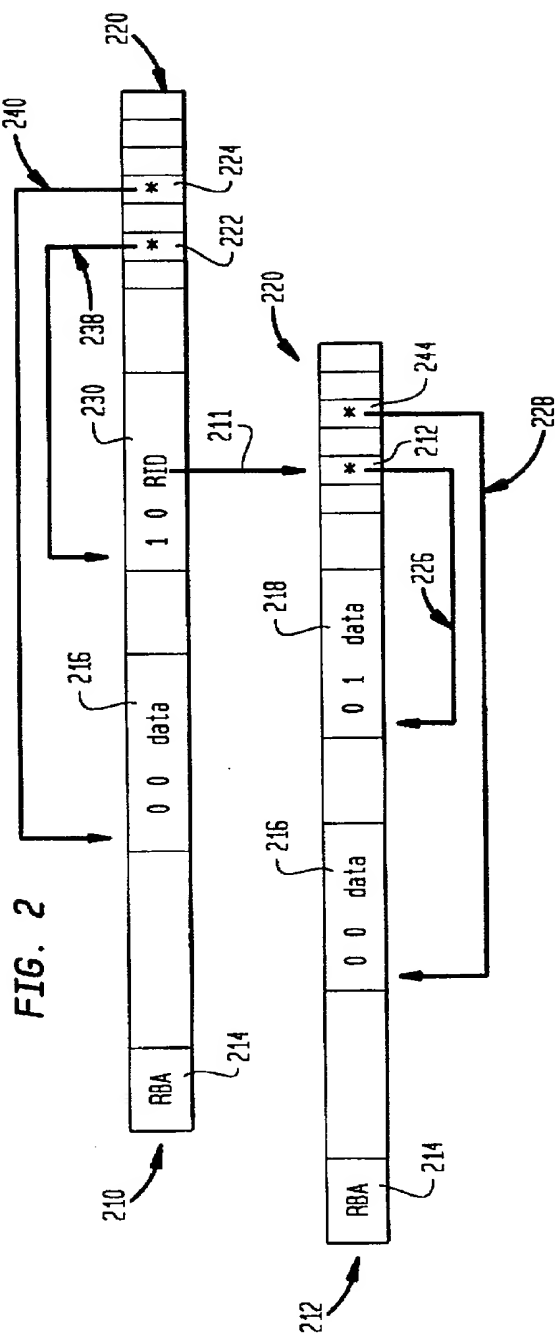


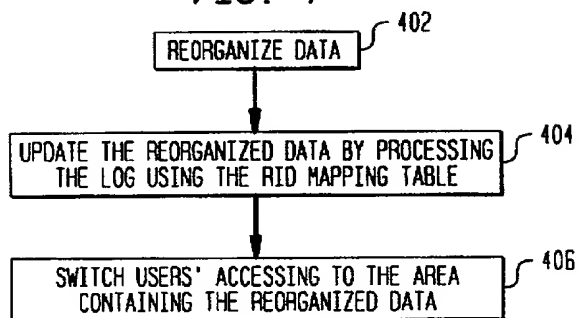
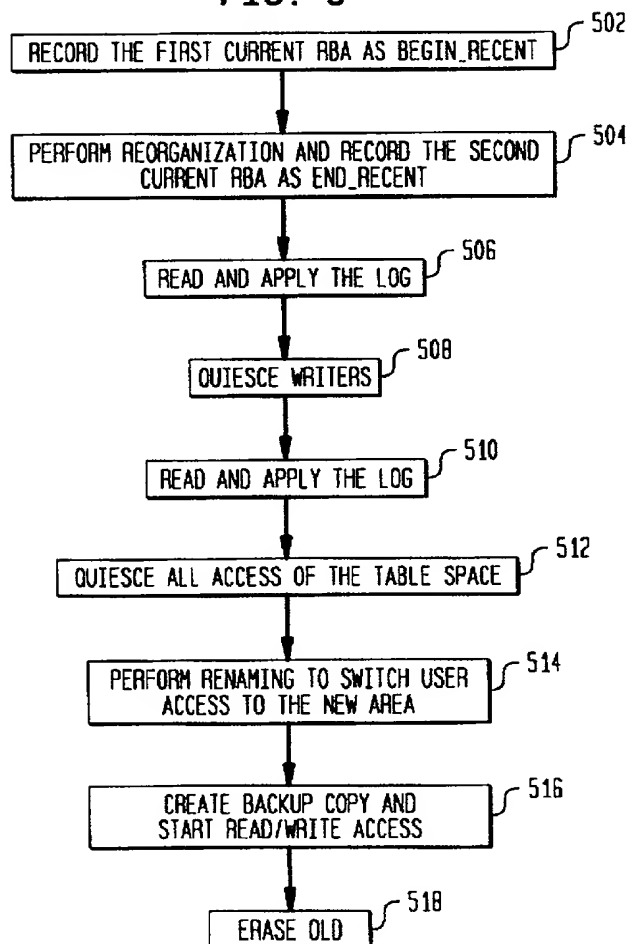

FIG. 4**FIG. 5**

FIG. 6

610



TYPE	SOURCE_RID	TARGET_RID	RBA
CR	RID OF OLD REGULAR OR OVERFLOW	RID OF NEW REGULAR OR POINTER	RBA
CE	RID OF OLD REGULAR OR OVERFLOW	ESTIMATED RID OF NEW REGULAR	RBA
P	RID OF OLD POINTER	(UNUSED)	RBA

FIG. 7

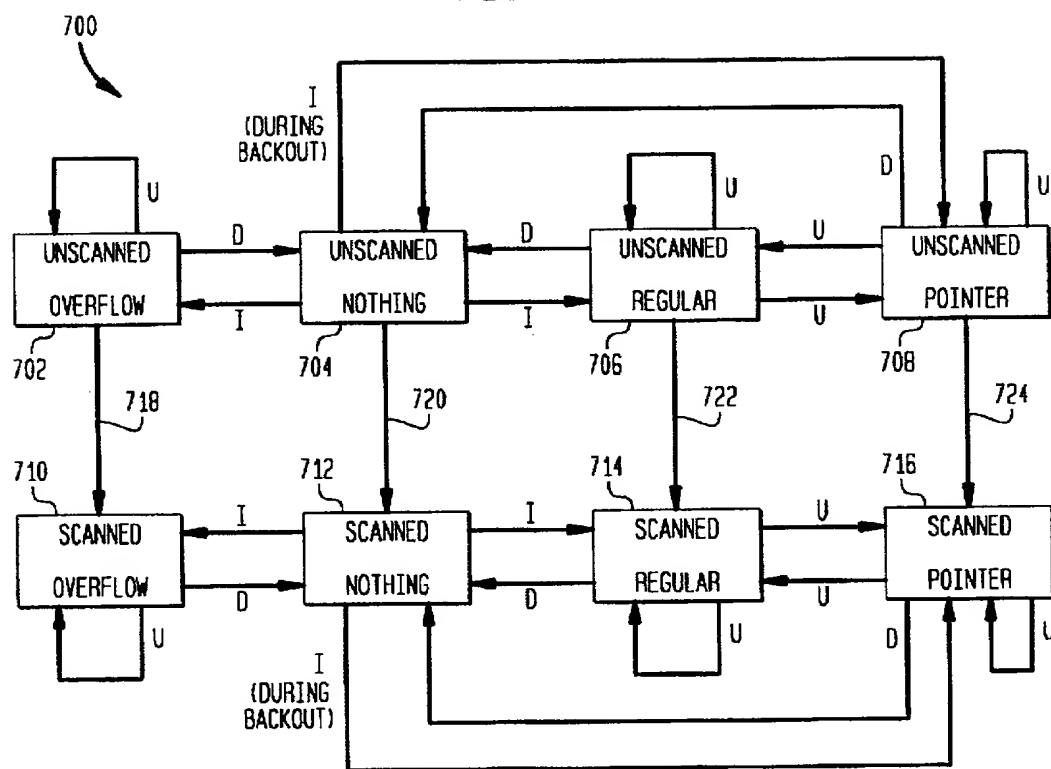
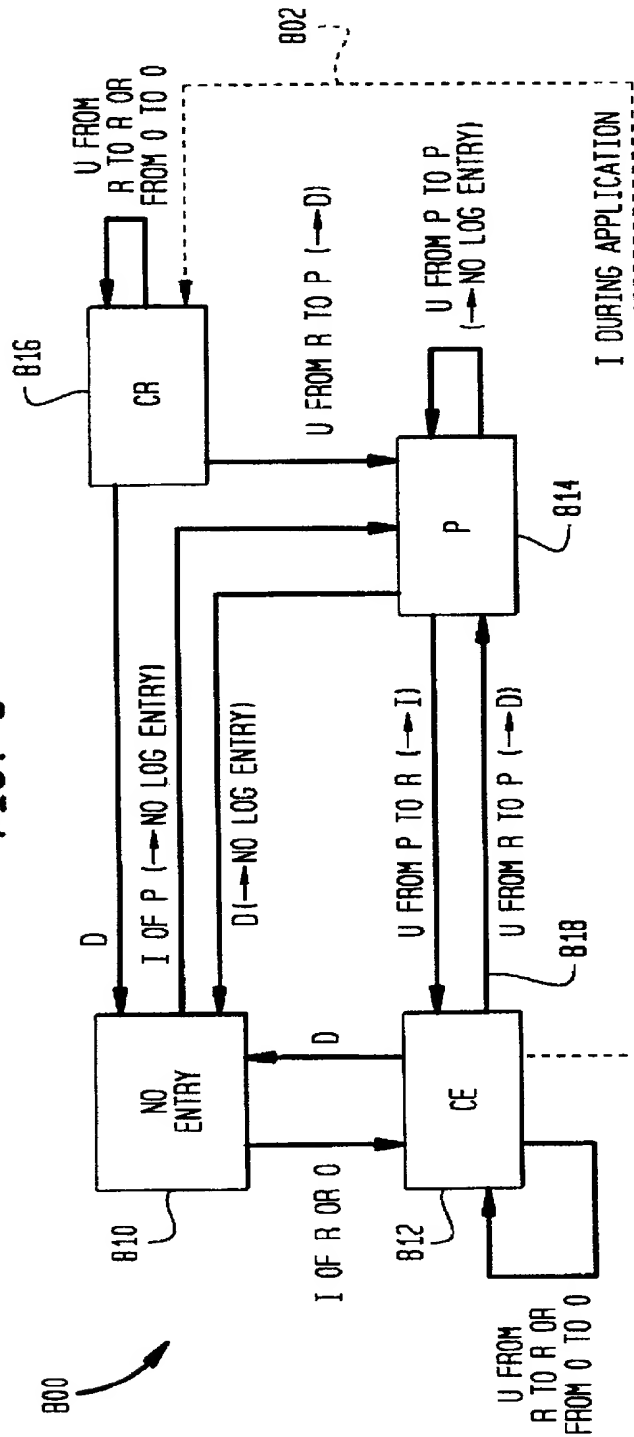


FIG. 8



INTERACTION BETWEEN APPLICATION OF A LOG AND MAINTENANCE OF A TABLE THAT MAPS RECORD IDENTIFIERS DURING ONLINE REORGANIZATION OF A DATABASE

This application is a continuation of application Ser. No. 08/366,564, filed Dec. 30, 1994, currently pending.

TECHNICAL FIELD

The invention relates to online reorganization of a database. More particularly, the invention relates to the interaction between the application of a log and maintenance of a table that maps record identifiers (RIDs) during online reorganization of the database.

BACKGROUND ART

Over time and with frequent use, databases often become disorganized. Accordingly, numerous attempts have been made to reorganize such databases. Reorganization of a database includes changing some aspect of the logical and/or physical arrangement of the database. Any database management system (DBMS) will require some type of reorganization. One type of reorganization involves restoration of clustering and removal of overflows. In particular, clustering relates to the storing of instances near each other if they meet certain criteria. One popular criterion is consecutive values of a key. Clustering is intended to reduce disk traffic for instances that a user often accesses in temporal proximity. However, a user's writing may fill data pages, decrease the amount of clustering, and degrade database performance. Accordingly, reorganization may restore performance.

Reorganization is a necessary feature in Database Management Systems (DBMSs). During most types of reorganization in a typical database, the area being reorganized is offline or only partially available. However, with a highly available database (a database that is to be fully available 24 hours per day, 7 days per week), it is undesirable to have the database go offline for significant periods. Database applications that require high availability include reservations, finance (especially global finance), process control, hospitals, police, and armed forces. Even for less essential applications, many database administrators prefer 24-hour availability. Moreover, reorganization of a very large database may require a long offline period for reorganization, which is usually longer than the maximum tolerable offline period. With the increasing sizes of databases, which may contain several terabytes or even petabytes of data, a user will likely experience even a longer offline period. Based on the increasing sizes of databases, as well as, increasing demands by users to have 24 hour database access, the need for online reorganization will very likely increase.

The considerations discussed above call for the ability to reorganize the database online (e.g., concurrently with usage or incrementally within users' transactions), so that users can read and write the database during most or all phases of reorganization. The desire to have this capability is well known. As the amount of information and dependence on computers both increase, the number of very large or highly available databases will increase. Therefore, the importance of online reorganization will increase.

One strategy for online reorganization is called fuzzy reorganization, which involves reorganization by copying. This type of reorganization involves a reorganizer (the process that performs the reorganization) that records a current relative byte address (RBA) of a log. An RBA is a

position in the log where a log entry can be written. At any time, the "current" RBA of the log is the position where the next log entry is written. An RBA is sometimes called a log sequence number (LSN). A log consists of a sequence of entries in a file (a region of storage), recording the changes that occur to a database. Then the reorganization copies data from an old (original) area for the table space to a new area for the table space, in reorganized form. Concurrently, users can use the DBMS's normal facilities to read and write the old area, and the DBMS uses its normal facilities to record the writing in a log. The reorganizer then reads the log and applies it to the new area to bring the new area up to date. Then, the reorganizer switches the users' accessing to the new area. In many PBMS's, however, each entry in the log identifies a record by the record's record identifier (RID). As an inherent part of reorganization the RIDs change. Therefore, when applying the log (which uses old RIDs) to the new area (which uses new RIDs), the problem of identification arises.

One way of overcoming this problem of identification is to have every record include a unique identifier that does not change during reorganization. However, it is often undesirable to have a restriction that each record must have a unique identifier that does not change during reorganization.

Another system describes garbage collection for persistent data by copying data and applying the log. See, for example, J. O'Toole et al., "Concurrent Compacting Garbage Collection of a Persistent Heap," Proc. 14th ACM Symp. Operating Syst. Principles, Dec. 1993 (Operating Syst. Review, SIGOPS, Vol. 27, No. 5), pp. 161-174. Each old record has a field that stores the address of the corresponding new record. Processing of the log uses this field to translate addresses in log entries. In a database context (which was not the context for this garbage collection) the technique of storing the address of the new record in the old record has serious disadvantages.

In particular, if a user deletes a record (and the DBMS generates a corresponding log entry) after the reorganizer has copied the record, then the reorganizer will eventually find the log entry, try to use the field to translate an address in the log entry from old to new, and apply the user's deletion in the new area. Between the user's deletion and the reorganizer's processing of the log entry, the DBMS might reuse the space that the deleted record occupied, so the new address may not be safely stored in the old (deleted) record. In addition, a data record is fairly large so the set of all data records can require many pages. Therefore, writing and reading the new addresses in the old data records can involve significant page input/output. Furthermore, storing the new record's address in the old record requires a shared lock while unloading the old record, an exclusive lock while reloading the new record (to write a new address in the old record), and a shared lock while processing the log (to translate the address). Therefore, this technique has a tendency to be slow and limits concurrency in the database. Furthermore, this technique may require extra space (which is permanent) in each data record for the address of the new record.

In contrast to fuzzy reorganization discussed above (which uses copying and a log), Reorganization can be performed in place (i.e., not by copying). One such strategy uses a table that maps RIDs to translate entries in the leaves of indexes. See, for example, E. Omiecinski et al., "Performance Analysis of a Concurrent File Reorganization Algorithm for Record Clustering," *IEEE Trans. Knowledge and Data Engin.*, Vol. 6, No. 2, Apr. 1994, pp. 248-257. Reorganization in place does not use the log because use of the

log is inappropriate here since there is only one copy of data. With this strategy for reorganization in place, each transaction by the user holds its locks until commitment of the transaction. However, this reduces the amount of concurrency in the database.

If each transaction by the user does not hold its locks until commitment of the transaction, reorganization in place is complex, especially if reorganization includes changing the assignment of records to pages, as in restoration of clustering. If a user scans a table space or index, and reorganization moves a record between the already-scanned area and the not-yet-scanned area, it must be assured that the user's scan processes the record exactly once, not twice or never. The feasibility of this approach has not yet been established. In addition, reorganization in place may cause more degradation of user performance, since this type of reorganization writes (instead of reading) the area that users access.

Furthermore, reorganization may be accomplished by offline reorganization of fine-grained partitions. Specifically, a partition of a table space can be a unit of offline reorganization or other utilities, during usage or offline reorganization of other partitions. With a fine enough granularity of partitioning, offline reorganization of a partition may be fast enough to approximate 24 hour availability. However, making the granularity fine can slow the routing of user accesses into the appropriate areas and increase the total space required for the partition's storage descriptors. It also increases the probability that areas of growth and areas of shrinkage will be in different partitions. This increase in the probability increases the likely variation among the partition's growth rates, thus increasing the total recommended amount of free space to reserve in the database. Also, offline reorganization (like some strategies for online reorganization) has a prerequisite period of quiescing of user activities.

What is needed is a system that is capable of reorganizing a database online or incrementally with minimal offline periods. By eliminating or minimizing the amount of time offline, a user may retain the ability to read and/or write to the database during all or most phases of reorganization. In particular, what is needed is online reorganization for restoration of clustering and removal of overflows.

Moreover, what is needed is a system for online reorganization with the following properties. Records that do not require a unique identifier which does not change during reorganization. The ability for users to insert, update, and delete data during reorganization. Lack of need for a user transaction to hold its locks until commitment of a transaction. Reorganization that does not significantly degrade user performance (e.g., by performing a large amount of locking). Reorganization that does not require permanent extra space in the area where user data is stored. An internal processing by the reorganization that limits the amount of page input/output.

Applicants have identified the ability to reorganize databases using a combination of a log application with the use of a mapping table. No prior work exists that combines log application with the use of a mapping table.

DISCLOSURE OF INVENTION

Reorganization of a Database Management System (DBMS) is disclosed. Reorganization is implemented by recording a first current relative byte address (RBA), which is a position within a log. Then, data is copied from an old area in the table space to a new area in the table space, in reorganized form. The new area is a new version of the table space after reorganization.

In the present invention, throughout reorganization a user maintains access to the DBMS's normal facilities to read and write to the old area. The DBMS uses its normal facilities to record writing, which occurs during reorganization, in a log. The reorganization in accordance with the present invention reads the log (that has been written to during reorganization) and applies the log to the new area to bring the new area up to date. However, after reading the log but before applying the logged writing, the reorganization of the present invention sorts the log entries by record identifier (RID). This sorting improves the locality of reference (and thus the speed) of log application. It also eases the detection (and omission during application) of a sequence of logged operations that has no net effect (e.g., insert . . . update . . . delete). Finally, at the end of reorganization, the user's access is switched from the old area to the new area.

In particular, the log identifies a record by the data record's record identifier (RID). As an inherent part of reorganization, RIDs change. Accordingly, since the log entries correspond to user writing of the old area, the log entries use the old RIDs. To apply a log entry to the new area, the record in the new area to which the entry should apply must be identified (i.e., the new RID). In the present invention, identification of the new record (by new RID) is done by maintaining a temporary table that maps between the old and new RIDs. The strategy uses this table to translate log entries before sorting them (by new RID) and applying them to the new area.

BRIEF DESCRIPTION OF DRAWINGS

The foregoing and other objects, features and advantages of the invention will be apparent from the following more detailed description of the invention, as illustrated in the accompanying drawings.

FIG. 1 shows an environment in which the present invention operates;

FIG. 2 shows exemplary file pages for a table of the present invention;

FIG. 3 shows exemplary index pages for a table of the present invention;

FIG. 4 shows an overview of the steps of online database reorganization system of the present invention;

FIG. 5 shows detailed steps of the online database reorganization system of the present invention;

FIG. 6 shows a mapping table of the present invention;

FIG. 7 shows a state transition diagram for a RID during user writing and the reorganizer's unloading; and

FIG. 8 shows a state transition diagram for an old RID's mapping table entry during processing of the log.

BEST MODE FOR CARRYING OUT THE INVENTION

An embodiment of the invention is now described with reference to the figure where like reference numbers indicate identical or functionally similar elements. Also in the figures, the left most digit of each reference number corresponds to the figure in which the reference number is first used. While specific configurations and arrangements are discussed, it should be understood that this is done for illustrative purposes only. A person skilled in the relevant art will recognize that other configurations and arrangements can be used without departing from the spirit and scope of the invention. It will be apparent to a person skilled in the relevant art that this invention can also be employed in a variety of other devices and applications.

Reorganization of a database means changing some aspect of the logical and/or physical arrangement of the database. Any DBMS will need some type of reorganization.

One type of reorganization involves restoration of clustering and removal of overflows. Clustering is the policy of storing instances near each other if they meet certain criteria. One popular criterion is consecutive values of a key. Clustering should improve performance by reducing disk traffic for instances that a user often accesses in temporal proximity. Of course, a user's writing can fill data pages, decrease the amount of clustering, and degrade performance. Reorganization can restore performance.

FIG. 1 illustrates an environment in which a preferred embodiment of the present invention operates. The preferred embodiment of the present invention operates on a computer platform 104. The computer platform 104 includes certain hardware units 112 including one or more central processing units (CPU) 116, a random access memory (RAM) 114, and an input/output (I/O) interface 118. The computer platform 104 includes an operating system 108, and may include microinstruction code 110. A database management system 103 (hereinafter DBMS 103), which may include online reorganization, uses operating system 108. Application programs can use DBMS 103 and operating system 108. Various peripheral components may be connected to the computer platform 104, such as a terminal 126, a data storage device 130, and a printing device 134.

DBMS 103, when executed, enables the computer platform 104 to perform the features of the present invention as discussed herein. Thus, DBMS 103 represents a controller of the computer platform 104.

DBMS 103 preferably represents a computer program or a library which resides (during run-time) in the main memory or RAM 114, and which is executed by the processors in the computer platform 104, such as CPU 116. The data maintained by DBMS 103 may be stored in, for example, data storage 130. Also, the computer program library associated with DBMS 103 may be stored in data storage 130, which may include, for example, a floppy disk or some other removable storage medium, which is read by the floppy drive or some other storage unit. The computer platform 104 may also be connected to a network. In this case, data may be retrieved from relational tables stored in storage devices in communication with computer platform 104 via the network.

The present invention may apply to a variety of storage structures where disorganization may arise and reorganization is needed. In particular, the present invention is discussed in the context of a set of storage structures for relational databases. Specifically, the storage structures used in IBM®'s DATABASE 2200, DB2®, and System R DBMSS are disclosed. (IBM, DATABASE 2, DB2 are trademarks of the International Business Machines Corp.) However, the present invention may also apply to other storage structures or data models.

In relational database management, data is stored in a two-dimensional data table, which may be stored in a table space. The relational DBMS may work with two data tables at the same time, relating the information or data through links established by a common column or field. In particular, the storage and retrieval of data occurs in the form of relational tables.

A table space of the present invention stores one or more tables. For purposes of this discussion, a single table will be referred to in a table space.

A table space contains file pages to store tables. FIGS. 2 and 3 show the structure of file pages and index pages,

respectively. In DB2, index pages are stored separately from the table space discussed herein.

Each file page shown generally at 210 and 212, contains zero or more data records (shown generally as 216, 218 and 230), which DBMS 103 allocates at the beginning of each of the pages 210 and 212. There are three types of data records, regular (216), pointer (230) and overflow (218). The end of file pages 210 and 212 contain an ID map shown generally at 220 which is an array of pointers (offsets of data records within the file page). Slot refers to the space (if any) to which an ID map entry points. Specifically, in file page 210, an entry (shown as "*") in pointer 224 points to a slot for regular data record and is shown generally as arrow 240; and an entry in pointer 222 points to a slot for pointer data record 230, and is shown generally as arrow 238. In file page 212, an entry in pointer 244 points to a slot for regular data record 216, and is shown generally as arrow 228; and an entry in pointer 242 points to a slot for overflow data record 218, and is shown generally as arrow 226. The header of each page includes the position in the log, i.e., the relative byte address (RBA 214) that was current when file page 210 or 212 was most recently written. Whenever users insert (I), update (U) or delete (D), DBMS 103 records the activity in the log and thus increases the current RBA.

In DB2 and several other DBMSS that use Structured Query Language (SQL), not every table has a unique key (a set of columns that identifies records). Therefore, file pages, indexes, and entries in the log cannot use a key for identification. Instead, they use a record's record identifier (RID), which consists of the record's page number and the offset of the record's entry within the ID map. A record's RID can change only during reorganization. In DB2, file pages, indexes and entries in the log always use a RID for identification even if a table does have a unique key.

On insertion of a record, or on growth by update of a variable-length column of a preexisting record, if the file page lacks enough contiguous free space, DBMS 103 compacts the file page to make its free space contiguous. (Insertion, deletion and update are discussed below.) If compaction does not produce enough space, the data goes onto another page. On insertion, the new data's RID denotes that other page (not shown). On growth of existing data, the existing data's RID still denotes the original page 210, but the existing data record on the original page 210 becomes a pointer data record 230 (a record containing the RID of the new overflow data record 218 on file page 212, which contains the actual data). Data records that do not involve overflow are regular data records (216). The header of data records 216, 218 and 236 contain bits (i.e., 0 or 1) to indicate whether the record is, for example, a pointer or an overflow.

Referring now to FIG. 3, an index is shown generally at 300. Index 300 contains a hierarchy of index pages 310, 312 and 14. There are two kinds of index pages: leaf pages and nonleaf pages. Every index contains both kinds of pages. The leaves are the bottom of the hierarchy and everything else is nonleaf. As shown, index leaf pages are shown generally at 312 and 314, and a nonleaf page is shown generally at 310. Each entry on leaf pages 312 and 314 contains a key value and a list of RIDs whose records have that key value; DBMS 103 can sort the list by RID. For each table, the database designer declares at most one index as a clustering index. In offline reorganization (and, whenever possible, in subsequent insertions), the assignment of data records to file pages reflects the data record's logical order in the clustering index. This clustering speeds some queries. The database designer can declare the clustering index to be a partitioning index; i.e., DBMS 103 divides the table into

partitions (an optional subdivision of a table) according to values of the indexed key.

Data can become disorganized when free space becomes unevenly distributed among the areas of a table space. After subsequent insertions, the order of data instances no longer reflects the clustering index. This slows some queries. In addition, data can become disorganized when variable-length data grows too large to fit on its original page, and DBMS 103 moves the excess data to a new page and makes the original record a pointer data record. Indexes still point to the original record. This causes an extra page reference on some queries.

In the present invention, reorganization of the database is performed while a user maintains the use of the DBMS's normal facilities to read and write to the old area (pre-reorganization) of the table space, and DBMS 103 uses its normal facilities to record the writing in the log.

FIG. 4 shows an overview of the steps of the present invention. The table space and the associated indexes are reorganized at step 402. Then the reorganized data is updated at step 404 by processing the log using a RID mapping table. Finally, a user's access is switched to the area containing the reorganized data at step 406. Step 402, 404 and 406 are discussed in detail below.

As discussed above, an entry in the log identifies a data record by the data record's RID. The present invention relates to the type of reorganization where the RIDs change during reorganization. In such reorganization, the log entries correspond to the user's writing to the old area and thus use the old RIDs. To apply a log entry to the new area, the record in the new area to which the log entry should be applied must be identified. In the present invention, the data record in the new area is identified by maintaining a temporary table that maps between the old and new RIDs. After this the table is used to translate the old RID to the new RID. Then the log entries are sorted and applied to the new area.

The interaction between processing of the log and maintenance of the mapping table of the present invention will now be discussed. In particular, the mapping table is updated (for each log entry) to reflect (1) a state (e.g., regular, overflow or pointer as discussed below) of the data record before processing of the log entry and (2) the type of log entry. The types of log entries may include insertions, deletions and updates. For a log entry that represents an insertion, this updating includes the use of an estimated new RID (as a basis for sorting) and eventual translation of estimated new RIDs to actual new RIDs. Estimated RIDs are discussed in greater detail below.

Referring now to FIG. 5, the steps of reorganization of the present invention are shown. The reorganization can apply to a partition of a table space or to an entire table space.

At step 502, the first current RBA for the log is recorded and stored as a variable called BEGIN—RECENT. The variable is part of the reorganizer's storage area. Users have read/write access to the old area during this step 502.

Next, at step 504, a reorganization (unload, sort, and reload) is performed by the reorganizer directing the reorganized version of the data record into a new area. During step 504, a user retains read/write access to the old area. The reorganizer maintains a temporary mapping table (discussed in greater detail below and shown in FIG. 6) that maps between the old and new RIDs for the data records. When the reorganizer unloads a record's data from the old area, it also "unloads" the old RID, not just the record's data. When reorganizer reloads into the new area, it reloads just the record's data, not a RID, but it also inserts an entry con-

taining the old RID and the new RID into the mapping table. Reloading also involves reconstructing indexes. If the entire table space is reorganized, all the indexes are reconstructed, that is, new copies of all the indexes are created. If a partition of the table space is reorganized, a partition of the clustering index is reconstructed, and a copy of a subset of each nonclustering index is created; the subset corresponds to the data records in the partition being reorganized. At the end of step 504, a second current RBA is recorded as a variable called END—RECENT.

Next, at step 506, a subset of the log is read. This subset consists of the entries from BEGIN—RECENT through END—RECENT—1. The log entries (for data) are applied to the new area of the table space or partition. This log processing will reflect user's writing that occurred during the previous step 504 or the previous iteration of this step 506. A log entry contains (among other things) an RBA, an old RID, and an operation, such as insertion or deletion. Log processing uses the temporary table that maps old and new RIDs. The log processing step 506 affects the indexes by modifying indexes to reflect the modifications to the data. For example, if a record is inserted in a table, and the table has indexes, then the indexes are modified to include the RID of the newly inserted index.

During step 506, a user has read/write access to the old area of the table space. At the end of an iteration of step 506 a comparison is made. In particular, (current RBA—END—RECENT), which is the amount that the next iteration would process, is compared to (END—RECENT—BEGIN—RECENT), which is the amount that this iteration processed. If the next iteration would process at least as much as the present one processed, then the reorganizer's reading of the log is not catching up to user's writing of the log. Accordingly, an action is taken to solve this problem. A parameter of the reorganization command determines the action that should be implemented to allow the reorganizer's reading of the log to catch up to the user's writing of the log. The possible actions are quiescing writers of the table space or partition, increasing the scheduling priority of reorganization, aborting reorganization, and asking the operator to choose one of the above actions. If the reorganization is not aborted, BEGIN—RECENT is set to END—RECENT; and END—RECENT is set to the second current RBA. Then, if END—RECENT—BEGIN—RECENT exceeds a size (another parameter), and the number of iterations so far is less than a limit (another parameter), then another iteration of log processing step 506 is begun. Otherwise, the next step 508 is proceeded to. The purpose of comparing the above with the size and limit parameters is to limit the number of iterations of step 506.

At step 508, writing to the table space or partition is quiesced, if the previous step 506 did not already quiesce this writing. Then, END—RECENT is set to the third current RBA for the log. A user may continue to have read-only access to the old area during step 508.

At step 510, the log is processed again, from BEGIN—RECENT through END—RECENT—1. This last step of processing of the log is necessary only to handle writing that was in progress when (or that began after) the previous step 506 of processing of the log finished reading the log. A user has read-only access to the old area during step 510.

At step 512, all access of the table space or partition is quiesced.

At step 514, a renaming (i.e., change the mapping from logical to physical) is performed so that all future access to the table space will use the new area. Similarly, renaming is

performed so that all future access of the indexes or index partition, which were reconstructed as discussed above, will use the reconstructed versions. If only a partition is being reorganized, RIDs are corrected for this partition in any nonclustering indexes (in place, not by copying) as follows. In particular, for each nonclustering index, for each key value, the old RIDs for this partition are replaced by the new RIDs (in the constructed subset of the index discussed above in step 504). For each key value, the old RIDs for this partition are contiguous only if the index's definition specified that the RIDs in the leaves will be sorted (and thus grouped by partition). The correction of RIDs for a sorted index is faster than for an unsorted index. With one exception, a user has no access to the old area during step 512. The exception is that during the correction of nonclustering indexes (for reorganizing a partition), queries that read just the index key values but not the RIDs can be allowed.

At step 516, a backup copy of the new table space or partition (as a basis for future recoverability) is created. Read/write access to the new area of the table space or partition is started. Possible sequences for this step 614 include: (1) start read-only access, create a backup copy while allowing read-only access, and then start read/write access (after the backup copying completes); or (2) start the creation of a backup copy via a facility that allows concurrent writing, and start read/write access as soon as the backup copying begins (instead of waiting for the backup copying to complete); or (3) create a backup copy during reorganization step 504, append translated log entries to the original log in steps 506 and 510, and start read/write access immediately in step 516.

At step 518, the old area, the old versions of reconstructed indexes, the copies of subsets of nonclustering indexes, and the mapping table are erased.

In accordance with the present invention, reading and writing during almost all activities in the reorganization, including the log processing step 506, are possible. Subsequent steps involve a period of read-only access (steps 508 and 510) and a period of no access (steps 512 and 514). The relatively short steps of read-only access (steps 508 and 510) or no access (steps 512 and 514) occur after processing of most of the logged writing.

To reorganize just a partition, step 514, discussed above, brings the partition completely offline during an expensive operation of changing entries in nonclustering indexes. As an alternative for changing entries in nonclustering indexes, in step 504, a copy of all (not only a subset) of each nonclustering index is constructed; in step 506 and 510, for each nonclustering index, the translated log entries are applied to the entries in those indexes for this partition (as discussed above), and also the untranslated log entries are applied to the entries in those indexes for the other partitions; and in step 514, all access to those indexes (even for the other partitions) is quiesced, and the indexes are replaced by the new copies of the indexes.

The advantage of this alternative is that it greatly shortens the amount of time in step 514, when this partition is offline. However, although the amount of time is shortened, this alternative requires briefly quiescing of all access to all the nonclustering indexes (even for the other partitions).
Structure of the Mapping Table

Referring now to FIG. 6, the mapping table is shown generally at 610. Mapping table 610 may be a database table or a special structure. Mapping table 610 includes columns and rows. The columns include, from left to right, TYPE,

SOURCE—RID, TARGET—RID, and RBA. The TYPE column actually contains numbers (or single characters), but table 610 uses symbols for illustrative purposes. The first character of the symbol relates to the old RID record. In particular, character C means that the old RID's record contains columns of data, and P means that the old RID's record is a pointer. The second character (if any) of the symbol also relates to the RID and has the following possible values and meanings. The character R means that the TARGE—RID is the actual RID of a new regular data record or pointer data record. The character E means that TARGET—RID is the estimated RID of a new record that will be inserted later when the log is applied. The determination of the estimated RID will be discussed in detail below.

For all values of TYPE, the RBA initially contains the RBA of the old page that contained the old RID, as of the time when the reorganization step 504 unloads the file page. Thereafter, processing of a log entry will result in updating of the RBA column in the mapping table's entry for the record to which the log entry applies.

In mapping table 610, the SOURCE—RID has a unique index. This index does not need uniqueness, since the old RIDs are unique. However, a unique index is smaller and may be beneficial in a query optimization. The TARGET—RID has a nonunique index for the reasons discussed below.

In an alternative implementation of the mapping table, the column for SOURCE—RID may be eliminated. In such case, there will be an entry for each possible source RID (i.e., each possible combination of page number and record number within the page). Based on the table space's number of pages, the page size and the record size (or minimum record size for a variable-length table), the number of entries may be calculated. This alternative implementation advantageously uses less space for each mapping table entry and eliminates the need for an index on SOURCE—RID (instead, a calculated offset from the start of the mapping table is looked at), which eliminates the space needed for the index and the time needed for accessing the index. Nonetheless, with this alternative implementation, each possible source RID (even one whose slot is empty) still requires space in the mapping table for an entry; if the file grows due to extension, the mapping table must grow similarly; and for each possible source RID a mechanism is required to determine whether an entry logically exists. One way to determine whether an entry logically exists is to use a value of "N" in the TYPE column, but this requires initialization.

Behavior of a RID and its Slot

For each type of writing by a user (i.e., a high-level operation of insertion, update or deletion), DBMS 103 performs one, two or three low-level operations, and DBMS writes corresponding log entries as follows.

Insertion (I):

One insertion of a regular record.

Update (U):

If there was no overflow and still is no overflow: one update occurs from a regular data record to a regular data record.

If there was no overflow, but now there is overflow: then one update occurs from a regular data record to pointer data record and one insertion of an overflow record occurs.

If there was an overflow, and the new data fits on the overflow's page: then one update occurs from overflow data record to overflow data record.

If there was an overflow, and the new data is too large for the overflow's page:

if there is now room on the pointer's page: then one update occurs from pointer data record to regular data record, and one deletion of an overflow record occurs; or

if there is still no room on the pointer's page: then one update occurs from pointer data record to pointer data record, one deletion of an overflow record occurs, and one insertion of an overflow record occurs.

Deletion (D):

If there is no overflow: then one deletion of a regular data record occurs.

If there is overflow: then two deletions occur, one deletion of a pointer data record and one deletion of an overflow data record.

During a back out of a transaction (i.e., cancellation of the effects of changes made by a transaction), DBMS 103 performs operations (i.e. I, U, D) and writes corresponding compensation log entries to reverse the original operations. A compensation log entry represents the operations that DBMS 103 performs in a backout. For example, to back out a deletion when there was overflow, two insertions are used, in particular, a pointer data record and an overflow data record.

Referring now to FIG. 7 a state transition diagram is shown generally at 700. Diagram 700 represents a state transition for a RID during writing by users (discussed above) and unloading by the reorganizer (discussed below) during reorganization step 504. The states indicate whether reorganizer has scanned the RID yet during the unloading part of reorganization step 504; and what the RID's slot contains (an overflow data record, nothing, a regular data record, or a pointer data record).

Four states, 702, 704, 706 and 708, include the four possible initial states (before the unloading begins). States 702, 704, 706 and 708 are all unscanned states. Specifically, state 702 is unscanned overflow data record, state 704 is unscanned nothing (discussed below), state 706 is unscanned regular data record and state 708 is unscanned pointer data record. Four states 710, 712, 714 and 716 include the four possible final states (after the unloading finishes). States 710, 712, 714 and 716 are all scanned states. Specifically, state 710 is scanned overflow data record, state 712 is scanned nothing, state 714 is scanned regular data record and 716 is scanned pointer data record. The transitions between states 702, 704, 706, 708, and the transitions among states 710, 712, 714 and 716, are shown generally as I, D and U. The transitions represent Insertion (I), Deletion (D), Update (U). The transitions shown generally as 718, 720, 722, and 724, represent the reorganizer's scan of that RID data (during unloading part of the reorganization step 504) by scanning the table space sequentially and unloading the data records. For each RID in the table space, the RID is a state. During the scan for transition 718, the overflow data record is scanned and the overflow data record is unloaded. Later, during reloading of the overflow data record, a CR entry is inserted into mapping table 610. During the scan for transition 720, nothing is scanned and nothing is done. During the scan for transition 722, the regular data record is scanned and the scanned regular data is unloaded (step 506). Later during reloading, a CR entry is inserted in mapping table 610. During the scan for transitional 724, the pointer data record is scanned. Then, a P entry is inserted into mapping table 610.

Unloading and Reloading within the Reorganization Step

The reorganizer unloads data by scanning the table space sequentially and unloading data records. The reorganizer then sorts the data by using the clustering key. Unloading is

performed as discussed briefly above. In particular, when a regular data record or an overflow data record (in the old area) is scanned; the data, the old RID and the RBA of the regular data record's old page or overflow data record's old page are unloaded. When a pointer record is scanned, as briefly discussed above, a P entry (including values for all the appropriate columns) is added to mapping table 610 and the pointer is not followed. Unloading an overflow data record at the same time the overflow data record's page is scanned, is more efficient than unloading the overflow data record when the pointer data record's page is scanned.

When a regular or overflow data record is reloaded (in the new area), a CR entry is added to mapping table 610, using the old RID and RBA from the unloaded old regular or overflow data record in the unload file and the new RID from the new area. Even if the old data record overflowed in the old area, the new data record will not overflow in the new area. Only the later processing of an update found in the log may cause an overflow in the new area. The order of adding CR entries to the mapping table is the order of new RIDs.

Instead of scanning the table space, an alternative is to scan the most recent backup copy of the table space. This alternative eliminates the need of reorganizer to access and lock the table space. However, this alternative requires the reorganizer to process additional log entries, in particular, all the entries since the backup copy was made.

The later processing of log entries will include access to the mapping table by old RID. To make this access more efficient (largely sequential instead of random), the log entries will be sorted by old RID before accessing the mapping table. Although user's writing has reduced the degree of clustering of the table space, there will be a correlation between the order of the record (and thus mapping table entries') old RIDs and the order of their new RIDs. Therefore, the order of the log entries (sorted by old RID) should approximate or match the order of the mapping table entries (added by new RID, as described above, or sorted by old RID, as described below). Therefore, access to the mapping table will be approximately sequential.

As CR entries are added to the mapping table, the average difference between the old and new RIDs is calculated. This average indicates the degree of clustering of the old area and thus the correlation between the order of the old RIDs and the order of the new RIDs. If the average exceeds a threshold, and the number of log entries since reorganization was begun exceeds a threshold, then the mapping table is sorted by old RID at the end of this step, to speed the later access to the mapping table during processing of the log. The comparisons to the thresholds determine whether this sorting is worthwhile.

Processing the Log

Steps 506 and 510 in the reorganization of the present invention process the log. Log application has more locality of reference and thus is faster if the log entries are sorted by RID before applying them. The sorting also eases the detection (and omission during application) of a sequence of logged operations that has no net effect (e.g., insert . . . update . . . delete). This omission covers log entries which no longer have the appropriate entries in the mapping table. The sorting should use the new RID. As discussed earlier, sorting the log entries by old RID should speed the access to the mapping table for translation from old to new RIDs.

In the present invention, processing of the log has the following phases:

1. Sorting by old RID: Sort pointers to the log entries by old RID.
2. Translation: Copy the subset of the log that applies to the area being reorganized and whose RBA's are from

BEGIN—RECENT to END—RECENT—1 and translate the RID of each log entry from old to new.

3. Sorting by new RID: Sort pointers to the translated log entries by new RID. This phase may be combined with the previous phase.

4. Application: Apply the translated, sorted log entries to the new data area.

For an insertion, the new RID is not known until the insertion is actually performed, which will occur after the sorting by new RID. Therefore, the translation phase (which occurs before the sorting by new RID) calculates an estimated new RID for the inserted record, based on the page numbers for records that have similar values for the clustering key and that already exist in the new area. It is efficient but not logically necessary for the estimated new RID to be close to the eventual actual new RID.

Referring now to FIG. 8, a state transition diagram for an old RID's mapping table entry (if any) during log processing is shown generally at 800. Each state represents a type of entry (or no entry) in the mapping table 610. Diagram 800 includes state 810 as a "no entry" state, state 812 as a CE state (CE is discussed above), state 814 as a P state (p is discussed above) and state 816 as a CR state (CR is discussed above). The dashed transition 802 represents the processing of an insertion log entry during the phase for application. The transitions shown by solid lines represent the processing of applicable log entries during the phase for translation. Each solid line transition is labeled with the type of log entry (I, D, or U). For an insertion, the type of old record (R for regular, P for pointer, or O for overflow) is shown. For an update, the type of old record is also shown before and after the update. If the translation changes the type of log entry, the changed type of log entry appears after an arrow, both of which are shown in parentheses. For example, a transition 818 represents a log entry that updates (U) the old data record from a regular data record (R) to a pointer data record (P) when the old RID has a CE entry in the mapping table 610. In this transition, the log entry is changed from update to deletion (D), and the mapping table entry is changed from CE to P. The sections below explain how to apply the log; the explanations cover all the transitions in the diagram.

1. Phase for Sorting by Old RID

Construct pointers to the log entries and sort the pointers by old RIDs of the log entries. This sorting speeds the later accessing of the mapping table. The set of pointers could be arranged as an index; this is an implementation decision.

2. Phase for Translation

Scan the set of pointers to log entries, which is now sorted by old RID. This phase will modify RIDs in log entries. Reorganization must not modify the original log, since possible later recovery to a point in time might need the original log. Therefore, the relevant part of the log is copied as part of the translation phase, and the RIDs in the copy will be modified. As discussed hereinafter, most references to the log actually refer to the copy of the log.

A DBMS's log application (as part of recovery) ignores a log entry if its RBA is less than or equal to the RBA of the page to which the log entry refers. This behavior handles log entries that are inapplicable to what has been unloaded. Such a situation can occur if writing and the associated log entry occur after unloading begins but before the unloading reaches the record. The sequence of events may be as follows:

1. A RID's slot is empty when unloading begins.
2. A user inserts a record there, and DBMS 103 appends an insertion entry for that RID into the log.

3. The unloading reaches that slot and reads the inserted record.

4. During log application, the log entry is found. Applying the log entry for insertion would not make sense, since the unloading has already read the inserted record. Therefore, this log entry is ignored.

Similarly, the translation phase refrains from copying (effectively ignores) an inapplicable log entry.

The next three sections describe how to translate insertion, update, and deletion entries found in the log.

Translation of an Insertion Entry Found in the Log

If the old RID has an entry in the mapping table, copying of the log entry is refrained from (the log entry is effectively ignored). Otherwise, if the insertion represents a regular or overflow record then calculate an estimated new RID. For unique identification during the later phase of application, this new RID must differ from all estimated or actual new RIDs now in the mapping table. Therefore, an index on TARGET RID is needed. ACE entry is inserted into the mapping table, using the old RID found in the log, the estimated new RID, and the RBA of the log entry. In the log entry, the RID is translated to the estimated new RID.

If the insertion represents a pointer record (which occurs only in a backup), then a P entry is inserted into the mapping table 610, using the old RID found in the log and the RBA of the log entry. Copying of the log entry is, again, refrained from. This behavior resembles the scanning of a pointer record during unloading.

Translation of an Update Entry Found in the Log

If the old RID does not have an entry in the mapping table 610, or the RBA in the log entry is less than or equal to the RBA in the mapping table 610, then copying the log entry is refrained from. Otherwise, if the log entry is an update from regular data record to regular data record or from overflow data record to overflow data record, then proceed as follows. If the old RID has a CR or CE entry in the mapping table 610 then, in the log entry, the RID is translated to the contents of TARGET—RID. In the mapping table 610, the RBA is changed to the log entry's RBA. If the old RID has a P entry in the mapping table then copying of the log entry is refrained from.

If the log entry is an update from pointer to pointer, then proceed as follows. If the old RID has a CR or CE entry in the mapping table 610 then refrain from copying the log entry. If the old RID has a P entry in mapping table 610 then change RBA to the log entry's RBA in the mapping table 610. Copying of the log entry is refrained from.

If the log entry is an update from regular to pointer, then proceed as follows. If the old RID has a CR or CE entry in mapping table 610 then the RID is translated to the contents of TARGET—RID in the log entry, and the type of log entry is changed from update to deletion. In mapping table 610, RBA is changed to the log entry's RBA, and TYPE is changed to P. This is done because the data (for this slot) has already been found in the old area or in the log. However, this slot (in the old area) has become a pointer. Therefore, the already found data is already known. Thus, a deletion entry is generated in the log. The real data, from the corresponding overflow in the old area, is retrieved when the RID is processed for the overflow in the old area. Since the new copy of the data corresponds to old regular records and old overflow records but not to old pointer records; an update from regular to pointer is treated similarly to a deletion. If the old RID has a P entry in the mapping table 610 then copying of the log entry is refrained from.

If the log entry is an update from pointer to regular data record then proceed as follows. If the old RID has a CR or

CE entry in the mapping table 610 then refrain from copying the log entry. If the old RID has a P entry in the mapping table 610, then as in an insertion, an estimated new RID is calculated. In the log entry, the RID is translated to the estimated new RID, and the type of log entry is changed from update to insertion. In the mapping table 610, TYPE is changed to CE, RBA is changed to the log entry's RBA, and TARGET—RID is changed to the estimated new RID. Since the new copy of the data corresponds to old regular records and old overflow records but not old pointer records, an update from pointer to regular data record is treated similarly to an insertion.

Translation of a Deletion Entry Found in the Log

If the old RID has no entry in the mapping table 610, or the RBA in the log entry is less than or equal to the RBA in the mapping table 610, refrain from copying the log entry. Otherwise, proceed as follows. If the old RID appears in a CR or CE entry in the mapping table 610, then translate the RID in the log entry to the contents of TARGET—RID, and delete the entry from the mapping table 610. If the old RID appears in a P entry in the mapping table 610, then refrain from copying the log entry, and delete the entry from the mapping table 610. The reason for this is that the log contains another entry to delete the corresponding old overflow, and eventually that other log entry, not this log entry, will be translated to delete the actual data.

Alternatives for Checking Log Entries

In the description so far, the translation phase of log processing compares the RBA of the log entry to the RBA in the mapping table's entry for the record to which the log entry applies. This phase also checks the mapping table for appropriateness of operations; e.g., updating a nonexistent record is inappropriate. These two checks are probably redundant in most DBMSs, so the implementor can choose among the following implementations: (1) compare the RBAs, and also check for appropriateness of operations; either type of checking can result in ignoring a log entry (as discussed above); (2) compare the RBAs, and also check for appropriateness of operations; the RBA comparison can result in ignoring a log entry; the appropriateness checking can result in abnormal termination of reorganization; any log entry with a good RBA should not contain an inappropriate operation; or (3) omit the RBAs from the mapping table 610, thus saving space and omitting the RBA comparison; the appropriateness checking can result in ignoring a log entry.

3. Phase for Sorting by New RID

Sort pointers to the translated log entries, using a major sort on RID and a minor sort on RBA. The major sort speeds the later actual processing of the log. The minor sort preserves the logical order of events. The set of pointers may be arranged as an index.

4. Phase for Application

Scan the set of pointers to log entries, which is now sorted by new RID. Log application is performed for each RID value in the log. These are new RID values (including estimated new RID values for inserted records). This is why an estimated new RID must differ from other estimated new RIDs and from real new RIDs that were already in the mapping table 610 when the new RID was estimated.

For each RID value in the log, the following is done.

1. Find all the log entries for that RID. They will be contiguous.

2. If there is at least one D log entry for that RID, perform these deletions of log entries:

if the first log entry is I (i.e., the slot is initially empty), delete the last D log entry and all the preceding log entries;

if the first log entry is D or U (i.e., the slot is initially occupied), keep the last D log entry, but delete all the preceding log entries.

These deletions omit log entries which no longer have the appropriate entries in the mapping table 610. These deletions also omit log entries whose effects would be nullified by a later D log entry. For example, the sequence I U U D has no net effect, so there is no need to waste the time to apply it.

3. If the preceding deletions deleted all the log entries for the RID, application for this RID is finished. If there are still log entries, apply them sequentially, as described below.

The next three sections describe how to apply insertion, update, and deletion entries found in the log.

Application of an Insertion Entry Found in the Log

The following is done in the processing of an insertion entry found in the log. Insert the record in the new area, and obtain its actual new RID. In the mapping table, find the CE entry whose TARGET—RID (estimated new RID) matches the log entry's estimated new RID. This is another reason why the index on TARGET—RID is unnecessary. In that entry in the mapping table, change the TYPE from CE to CR, and change the TARGET—RID from its estimated value to the actual value. In all the log entries for the current RID, change the RID from the estimated value to the actual value.

It is possible that the actual new RID will equal the estimated new RID of this record or of a different record. This is why the index on TARGET—RID is not unique.

For each new RID value in the log, log application is performed for all the log entries for that RID before the log application is performed for any other new RID value in the log. This is why it is not a problem if the actual new RID for an inserted record equals the estimated new RID for any inserted record.

Application of an Update Entry Found in the Log

The RID in the log entry identifies a regular record or a pointer record. The behavior of log application (in the new area) resembles the behavior of the DBMS's processing of a user's update. If there was no overflow and still is no overflow, update the regular record. If there was no overflow, but now there is overflow, update the regular record to become a pointer, and insert an overflow record. If there was an overflow, and the new data fits on the overflow's page, update the overflow record. If there was an overflow, and the new data is too large for the overflow's page then: (1) if there is now room on the pointer's page, update the pointer record to become a regular record, and delete the overflow record; if there is still not room on the pointer's page, update the pointer record, delete the overflow record, and insert an overflow record.

Application of a Deletion Entry Found in the Log

The RID in the log entry identifies a regular record or a pointer record. The behavior of log application (in the new area) resembles the behavior of the DBMS's processing of a user's deletion. If there is no overflow, delete the regular record. If there is overflow, delete the pointer record and the overflow record.

Type Of Scanning

The reorganizer scans the table space and then sorts by clustering key, instead of scanning the clustering index (which is already sorted). This decision has a correctness reason and a performance reason:

Correctness reason: In a scan of the clustering index, consider this sequence of events:

1. Initially, RID 3 contains a record whose clustering key value is Jones.

2. The reorganizer reaches the "J" portion of the clustering index, and it copies the Jones record from RID 3 in the old

area to RID 17 in the new. Now in the new area, RID 17 contains Jones.

3. A user updates the record, changing Jones to Smith. The log will record this as an update of RID 3 from Jones to Smith. Alternatively, users might delete Jones and insert Smith, and DBMS 103 might use the former RID of Jones for Smith.

4. The reorganizer reaches the "S" portion of the clustering index, and it copies the Smith record from RID 3 in the old area to RID 90 in the new. Now in the new area, RID 17 contains Jones, and RID 90 contains Smith. The mapping table 610 may now have two entries for old RID 3.

5. At the end of loading, the log is applied to the new area. Specifically, it is found that RID 3 was changed from Jones to Smith, and RID 17 in the new area is changed from Jones to Smith. Now in the new area, two RIDs (17 and 90) contain Smith. This is incorrect; there should be only one record for Smith. Some special handling would be necessary to prevent this error. The record for Jones in the new area is deleted (not updated).

Performance reason: Reorganization would probably only be performed if a table space has become unclustered. Studies have found that for offline reorganization of an unclustered table space, a table space scan and a sort together are usually faster than a scan of the clustering index, since an index scan for an unclustered table space involves many jumps between data pages.

Temporary Violation of Unique Indexes

During reorganization and log processing, a temporary violation of unique indexes might occur. Here is an example:

1. An early page of the table space contains a record whose unique key value is Jones.

2. The reorganizer unloads the record for Jones.

3. A user deletes Jones, and a user then inserts Jones in a later page of the table space.

4. The reorganizer unloads the second record for Jones.

5. The reorganizer reloads both records for Jones, thus temporarily causing a uniqueness violation.

6. The reorganizer eventually processes the log and deletes the first record for Jones, thus removing the uniqueness violation.

In the old area, the unique indexes enforce uniqueness. The set of values in the new area will eventually equal the set of values in the old area. Therefore, any such uniqueness violation is temporary.

In DB2, a nonunique index but not a unique index contains a field (for each key value) for a count of RIDs that have the key value. Therefore, this problem could not be solved by temporarily marking the unique indexes in the new area as nonunique.

Accordingly a solution is to always insert the data into the new table space. Try to insert entries into the unique indexes. If an insertion would violate uniqueness, instead save the RID in a special list. The list is probably short, since violation is probably rare. At the end of inserting data, again try to insert index entries for the saved RIDs.

Application of the log can cause modification of the special list. The last attempt to insert in the indexes, which occurs at the end of the last step of log processing, will always succeed, since the uniqueness violation is temporary. Omission of a Mapping Table if a Key is Unique

If the table that is being reorganized has a unique key, a complete mapping table is not needed. Here DBMS 103 must include the value of that key in each log entry, even for operations that do not involve that key (e.g., update of a different column). This will let reorganization use the unique key (instead of old RIDs) to identify log entries and apply them to new records.

The main features of the step that unloads, sorts by clustering key, and reloads are as follows.

(1) Unload regular and overflow records (and their old RIDs) by scanning the table space. (2) Sort, using a major sort by the unique key and minor sort by the old RID. When several records have the same value for the unique key, keep the one with the highest RID (i.e., the most recent one) and discard the others. The unique key will be used to identify the data records to which the log entries should apply. This sorting and discarding assure that the unique key really is unique. (3) If the clustering key is not the unique key, then sort by the clustering key, not the unique key. (4) Reload data into the new area, and build the indexes.

Processing of the log consists of the following phases

1. Omit the phase for sorting by old RID.

2. Copy the relevant part of the log, and construct a set of pointers to the log entries. This phase includes the following features:

If the last one or two log entries (low-level operations) and the first one or two entries in the next part of the log (which is not seen in this iteration of log processing) could represent parts of the same high-level user operation, defer those last one or two log entries to the next iteration. Specifically, these pairs or triples of log entries could represent the same user operation: (D pointer, D overflow), (I overflow, D overflow, U pointer to pointer), (D overflow, U pointer to regular), (I overflow, U regular to pointer), and (I pointer, I overflow). This deferral assures that the unique key stays unique.

Translate an update of the unique key into a deletion (whose key is the old value) and an insertion (whose key is the new value). This translation simplifies the processing of the log.

3. Sort the pointers by clustering key or new RID. Specifically:

If the clustering key is the unique key, use a major sort by clustering key and a minor sort by RBA. The resulting order approximates but does not perfectly match the RID order. A mapping table is not needed.

If the clustering key is not the unique key, sort by the unique key (to speed access to the unique index). Then use a major sort by the new RID (which is obtained from the unique index) and a minor sort by RBA. For an insertion, estimate a new RID (which must differ from all existing actual or estimated RIDs) and store it in a mapping table (which reflects only inserted records) or in a fake entry in the unique index. For a deletion of such an inserted record, delete the temporary table entry or fake index entry. Insertions could not be applied immediately while deferring updates and deletions, since a sequence like I U D I might occur for a value of the unique key.

The RBAs are unique (except for an update of the unique key, which were translated into a deletion and an insertion, which apply to different key values).

4. In the list of pointers, for each clustering key value (if the clustering key is unique) or each new RID (if the clustering key is not unique), do the following:

If there is at least one deletion log entry, discard all the log entries before the last deletion.

Apply the remaining log entries. Specifically, for a deletion or update, if the record exists, apply the log entry. If the record does not exist, discard the log entry. For an insertion, if the record exists, discard the log entry. If the record does not exist, apply the log entry. Also, for an insertion, if the clustering key is not the unique key, delete the temporary table entry or fake index entry.

In one embodiment, the present invention is a computer program product (such as a floppy disk, compact disk, etc.)

comprising a computer readable media having control logic recorded thereon. The control logic, when loaded into RAM 114 and executed by the CPU 116, enables the CPU to perform the operations described above. Accordingly, such control logic represents a controller, since it controls the CPU during execution.

While the invention has been particularly shown and described with reference to preferred embodiments thereof, it will be understood by those skilled in the art that (various changes) (the foregoing and other changes) in form and details may be made therein without departing from the spirit and scope of the invention.

Having thus described our invention, what we claim as new and desire to secure by Letters Patent is:

1. A program storage device readable by a machine, tangibly embodying a program of instructions executable by the machine to perform method steps for performing reorganization in a database management system (DBMS), said method steps comprising:

reorganizing a data record in an old area of a table space, wherein said data record has an old record identifier (RID), while read/write access to said old area is retained;

directing said reorganized version of said data record to a new area in said table space, wherein said reorganized version of said data record has a new RID, while read/write access to said old area is retained;

maintaining a mapping table that maps between said old RID and said new RID;

copying a subset of a log;

translating a log entry RID for a log entry to said new RID, wherein said log entry RID is said old RID; and

applying said translated log entry with said new RID to said reorganized version of said data record in said new area in said table space.

2. The program storage device according to claim 1, wherein said steps of reorganizing and directing comprise the steps of:

unloading said data record from said old area;

reorganizing said data record;

reloading said reorganized version of said data record, except said old RID, in said new area; and

inserting an entry containing said old RID and said new RID into said mapping table during said reloading step.

3. The program storage device according to claim 1, wherein when said log entry comprises a plurality of log entries, and said method further comprises the steps of:

sorting said log entries by said old RIDs, prior to said step of translating; and

sorting said log entries by said new RIDs, after said step of translating and prior to said step of applying.

4. The program storage device according to claim 1, wherein said step of translating comprises the step of:

calculating an estimated new RID.

5. The program storage device according to claim 1, wherein said step of maintaining a mapping table comprises the steps of:

updating said mapping table for each said log entry to reflect a state of said reorganized version of said data record, prior to said step of applying said log entry; and

updating said mapping table for each said log entry to reflect a type of said log entry.

6. The program storage device according to claim 1, wherein said step of applying said log entry comprises:

updating a relative byte address (RBA) column in said mapping table for said reorganized version of said data record.

7. A program storage device readable by a machine, tangibly embodying a program of instructions executable by the machine to perform method steps for reorganizing a database, said method steps comprising:

recording a first current relative byte address (RBA) for a log as a first variable, while read/write access to an old area of a table space is retained;

performing reorganization of a data record with an old record identifier (RID) in said old area of said table space such that a reorganized version of said data record has a new RID in a new area of said table space, while read/write access to said old area is retained;

storing a second current RBA for said log as a second variable;

reading said first variable;

reading said second variable;

applying a first log entry including a log entry RID, wherein said log entry RID is said old RID, to said reorganized version of said data record identified by said new RID in said new area by using a RID mapping table to translate said old RID to said new RID, while read/write access to said old area is retained;

recording a third current RBA for said log as a third variable;

comparing a next iteration, which is a difference between said third variable and said second variable, with a most recent iteration, which is a difference between said second variable and said first variable;

implementing an action to allow said reorganization to catch up if said next iteration is at least as much as said most recent iteration;

quiescing write access to said old area, while said second variable is set to said third variable;

applying a second log entry with a second log entry RID, wherein said second log entry RID is said old RID, to said reorganized version of said data record identified by said new RID, by using said RID mapping table to translate said old RID to said new RID, while read-only access is retained;

quiescing all access of said table space;

converting access from said old area to said new area;

creating a backup copy of said table space containing said new area;

starting read/write access to said table space containing said new area; and

erasing said old area and said RID mapping table.

8. The program storage device according to claim 7, wherein said step of performing reorganization comprises the steps of:

unloading said data record from said old area;

reorganizing said data record;

reloading said reorganized version of data record, except said old RID, in said new area of said table space; and

inserting an entry containing said old RID and said new RID into said mapping table during said reloading step.

9. The program storage device according to claim 7, wherein said first and second log entry each comprise a plurality of log entries, and prior to said steps of applying said first and second log entries, further performing the steps of:

sorting pointers to said first and second log entries by said old RIDs;

copying a subset of said first and second log entries; translating said old RID of each of said first and second log entries to said new RID; and

sorting pointers to said translated first and second log entries by said new RIDs.

10. The program storage device according to claim 9, wherein said step of translating by maintaining said mapping table further comprises the steps of:

updating said mapping table for each of said first and second log entries to reflect a state of said reorganized version of said data record; and

updating said mapping table for each of said first and second log entries to reflect a type of said log entry.

11. The program storage device according to claim 9, wherein said step of translating by maintaining said mapping table further comprises the step of:

updating a RBA column in said mapping table for said reorganized version of said data record.

12. A computer program product, comprising:

a computer usable medium having a computer readable program code means embodied in said medium for causing a database management system (DBMS) to be reorganized, the computer readable program code means comprising,

computer readable program code means for causing a computer to reorganize a data record in an old area of a table space, wherein said data record has an old record identifier (RID), while read/write access to said old area is retained;

computer readable program code means for causing a computer to direct said reorganized version of said data record to a new area in said table space, wherein said reorganized version of said data record has a new RID, while read/write access to said old area is retained;

computer readable program code means for causing a computer to maintain a mapping table that maps between said old RID and said new RID;

computer readable program code means for causing a computer to copy a subset of a log;

computer readable program code means for causing a computer to translate a log entry RID for a log entry to said new RID, wherein said log entry RID is said old RID; and

computer readable program code means for causing a computer to apply said translated log entry with said new RID to said reorganized version of said data record in said new area in said table space.

13. The computer program product according to claim 12, wherein said computer readable program code means for causing a computer to reorganize and direct comprises,

computer readable program code means for causing a computer to unload said data record from said old area;

computer readable program code means for causing a computer to reorganize said data record;

computer readable program code means for causing a computer to reload said reorganized version of said data record, except said old RID, in said new area; and

computer readable program code means for causing a computer to insert an entry containing said old RID and said new RID into said mapping table during said reloading step.

14. The computer program product according to claim 12, wherein when said log entry comprises a plurality of log entries, and said computer program product further comprises:

computer readable program code means for causing a computer to sort said log entries by said old RIDs, prior to said step of translating; and

computer readable program code means for causing a computer to sort said log entries by said new RIDs, after said step of translating and prior to said step of applying.

15. The computer program product according to claim 14, wherein said computer readable program code means for causing a computer to translate comprises:

computer readable program code means for causing a computer to calculate an estimated new RID.

16. The computer program product according to claim 12, wherein said computer readable program code means for causing a computer to maintain a mapping table comprises,

computer readable program code means for causing a computer to update said mapping table for each said log entry to reflect a state of said reorganized version of said data record, prior to said step of applying said log entry; and

computer readable program code means for causing a computer to update said mapping table for each said log entry to reflect a type of said log entry.

17. The computer program product according to claim 12, wherein said computer readable program code means for causing a computer to apply said log entry comprises:

computer readable program code means for causing a computer to update a relative byte address (RBA) column in said mapping table for said reorganized version of said data record.

18. A computer program product, comprising:

a computer usable medium having computer readable program code means embodied in said medium for causing a database management system (DBMS) to be reorganized, the computer readable program code means comprising,

computer readable program code means for causing a computer to record a first current relative byte address (RBA) for a log as a first variable, while read/write access to an old area of a table space is retained;

computer readable program code means for causing a computer to perform reorganization of a data record with an old record identifier (RID) in said old area of said table space such that a reorganized version of said data record has a new RID in a new area of said table space, while read/write access to said old area is retained;

computer readable program code means for causing a computer to store a second current RBA for said log as a second variable;

computer readable program code means for causing a computer to read said first variable;

a computer readable program code means for causing a computer to read said second variable;

computer readable program code means for causing a computer to apply a first log entry including a log entry RID, wherein said log entry RID is said old RID, to said reorganized version of said data record identified by said new RID in said new area by using a RID mapping table to translate said old RID to said new RID, while read/write access to said old area is retained;

computer readable program code means for causing a computer to record a third current RBA for said log as a third variable;

computer readable program code means for causing a computer to compare a next iteration, which is a

23

difference between said third variable and said second variable, with a most recent iteration, which is a difference between said second variable and said first variable;

computer readable program code means for causing a computer to implement an action to allow said reorganization to catch up if said next iteration is at least as much as said most recent iteration;

computer readable program code means for causing a computer to quiesce write access to said old area, while said second variable is set to said third variable;

computer readable program code means for causing a computer to apply a second log entry with a second log entry RID, wherein said second log entry RID is said old RID, to said reorganized version or said data record identified by said new RID, by using said RID mapping table to translate said old RID to said new RID, while read-only access is retained;

computer readable program code means for causing a computer to quiesce all access of said table space;

computer readable program code means for causing a computer to convert access from said old area to said new area;

computer readable program code means for causing a computer to create a backup copy of said table space containing said new area;

computer readable program code means for causing a computer to start read/write access to said table space containing said new area; and

computer readable program code means for causing a computer to erase said old area and said RID mapping table.

19. The computer program product according to claim 18, wherein said computer readable program code means for causing a computer to perform reorganization comprises,

computer readable program code means for causing a computer to unload said data record from said old area;

computer readable program code means for causing a computer to reorganize said data record; and

computer readable program code means for causing a computer to reload said reorganized version of data record, except said old RID, in said new area of said table space;

24

computer readable program code means for causing a computer to insert an entry containing said old RID and said new RID into said mapping table during said reloading step.

20. The computer program product according to claim 18, wherein said first and second log entry each comprise a plurality of log entries, and said article of manufacture further comprises:

computer readable program code means for causing a computer to sort pointers to said first and second log entries by said old RIDs;

computer readable program code means for causing a computer to copy a subset of said first and second log entries;

computer readable program code means for causing a computer to translate said old RID of each of said first and second log entries to said new RID; and

computer readable program code means for causing a computer to translate said old translated first and second log entries by said new RIDs.

21. The computer program product according to claim 20, wherein said computer readable program code means for causing a computer to translate by maintaining said mapping table further comprises:

computer readable program code means for causing a computer to update said mapping table for each of said first and second log entries to reflect a state of said reorganized version of said data record; and

computer readable program code means for causing a computer to update said mapping table for each of said first and second log entries to reflect a state of said log entry.

22. The computer program product according to claim 20, wherein said computer readable program code means for causing a computer to translate by maintaining said mapping table further comprises:

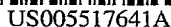
computer readable program code means for causing a computer to update a RBA column in said mapping table for said reorganized version of said data record.

* * * * *

ATTORNEY DOCKET NO.
063170.6564

PATENT APPLICATION
USSN 09/349,198

Appendix C: *Barry*



Barry et al.

[45] **Date of Patent:** **May 14, 1996**

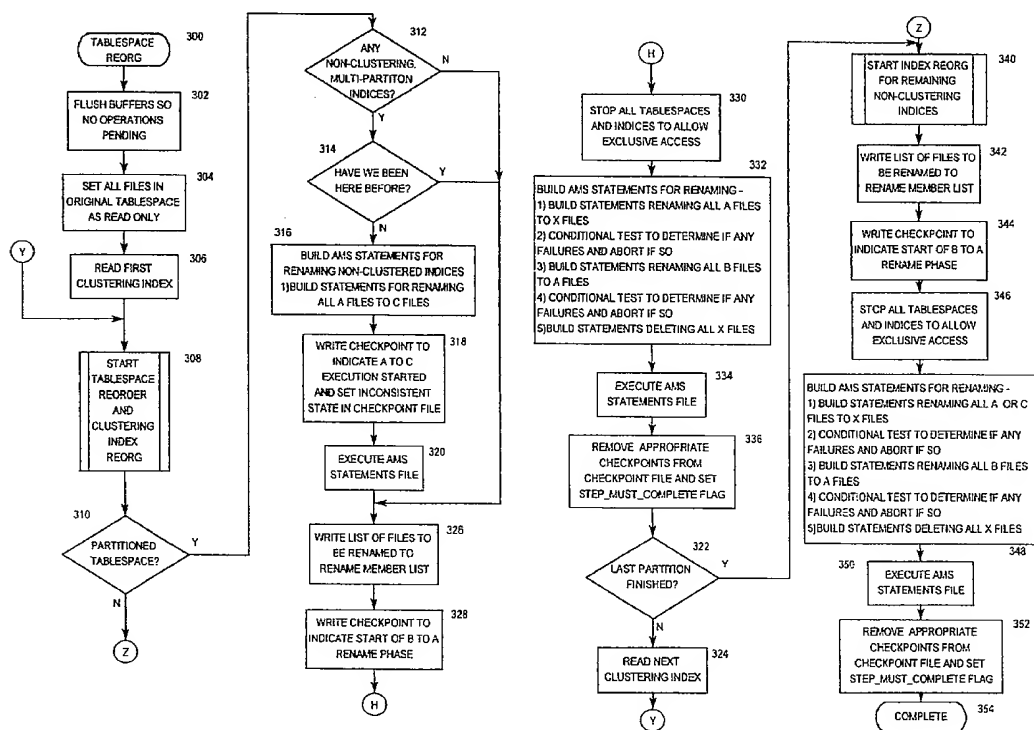
- [22] Filed: Dec. 7, 1993

[56] References Cited

4,679,139	7/1987	durbin	395/425
4,890,226	12/1989	Itoh	395/425
5,034,914	7/1991	Osterlund	395/425
5,204,958	4/1993	Cheng et al.	395/600

- [57]
- ABSTRACT**

23 Claims, 27 Drawing Sheets



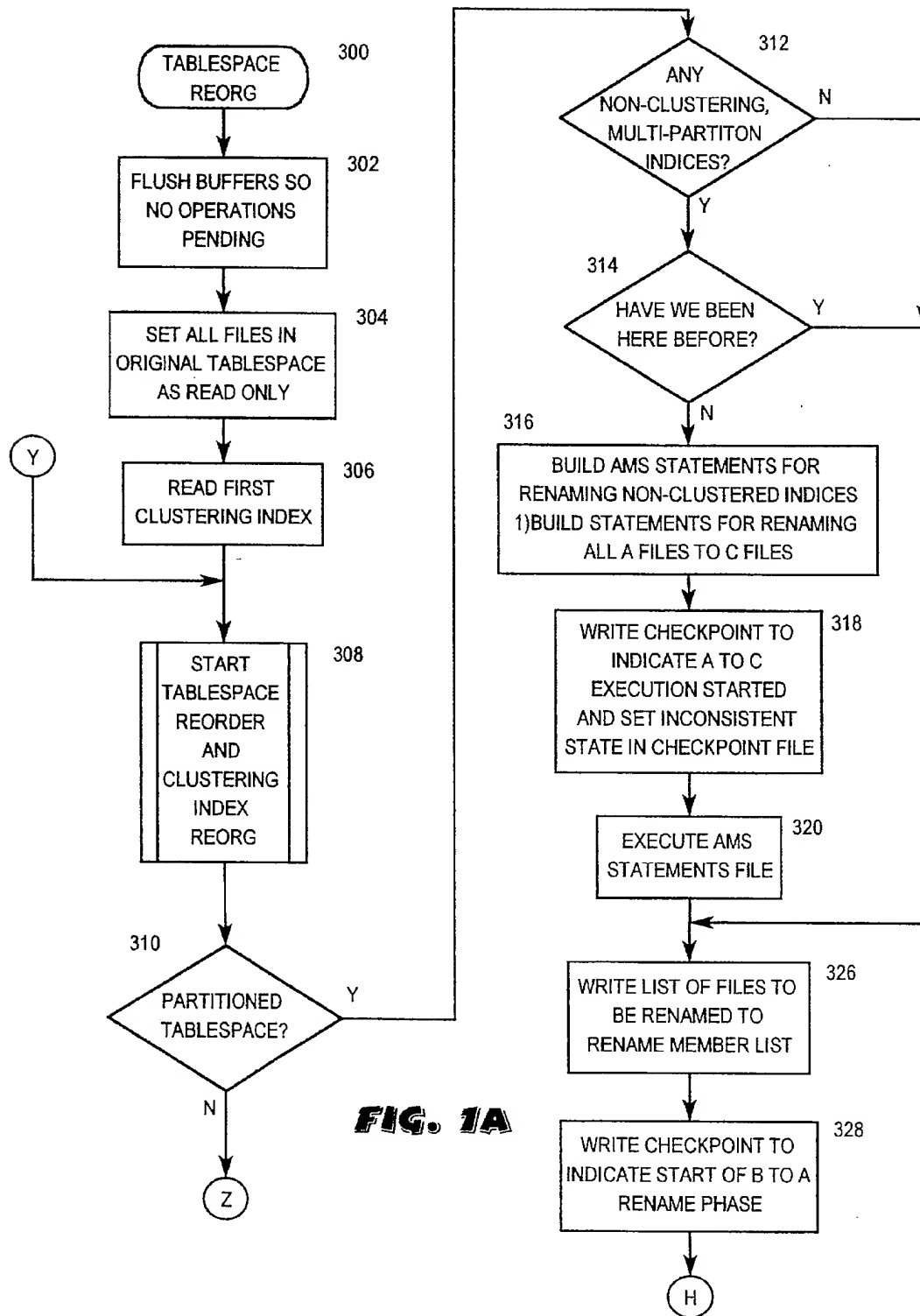
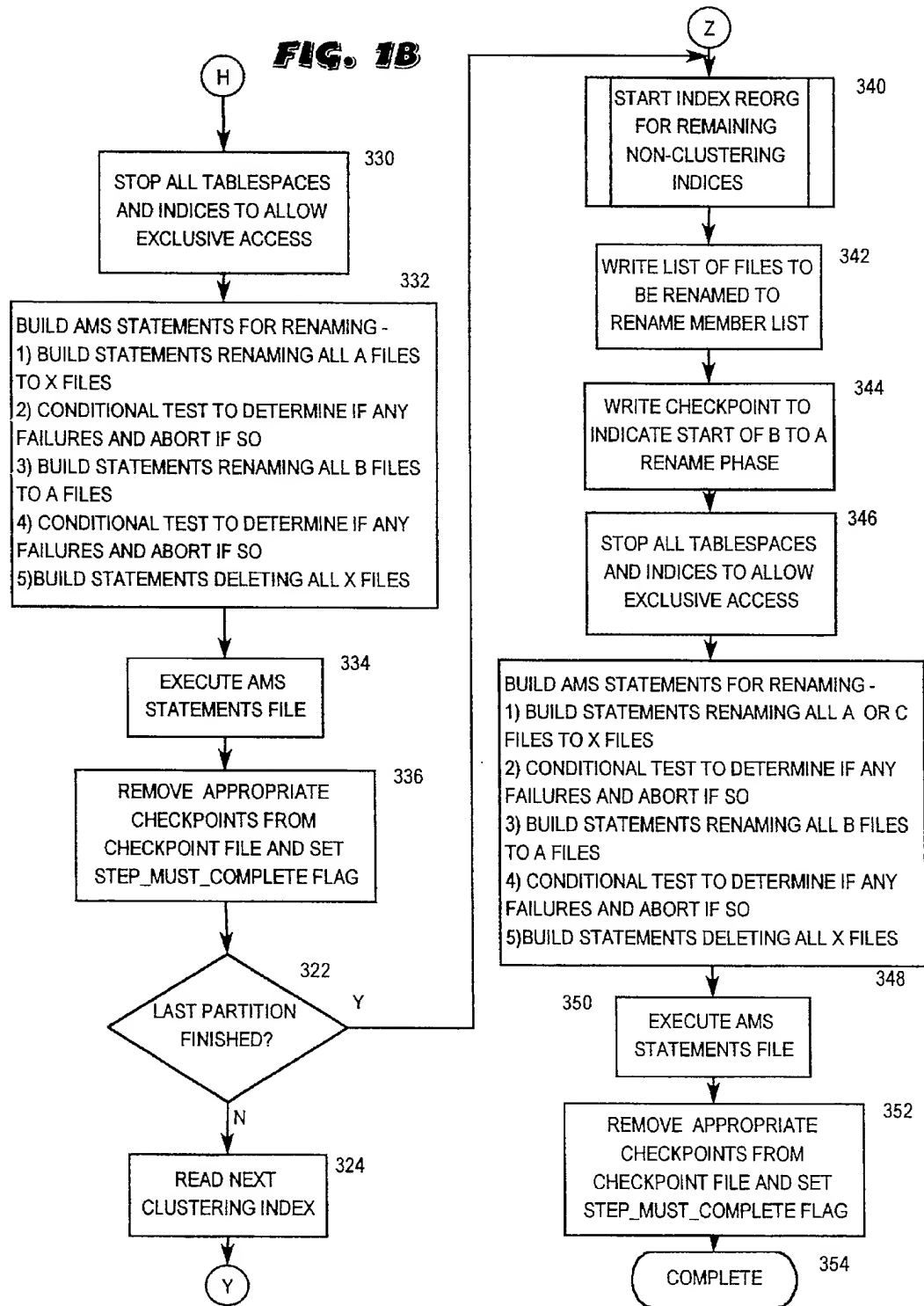
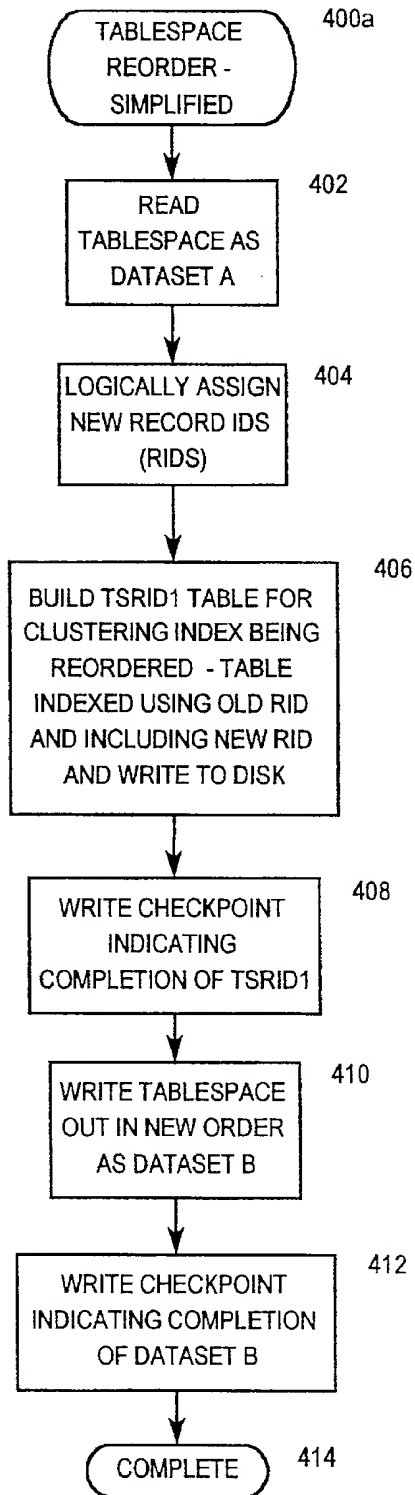


FIG. 1B

**FIG. 2**

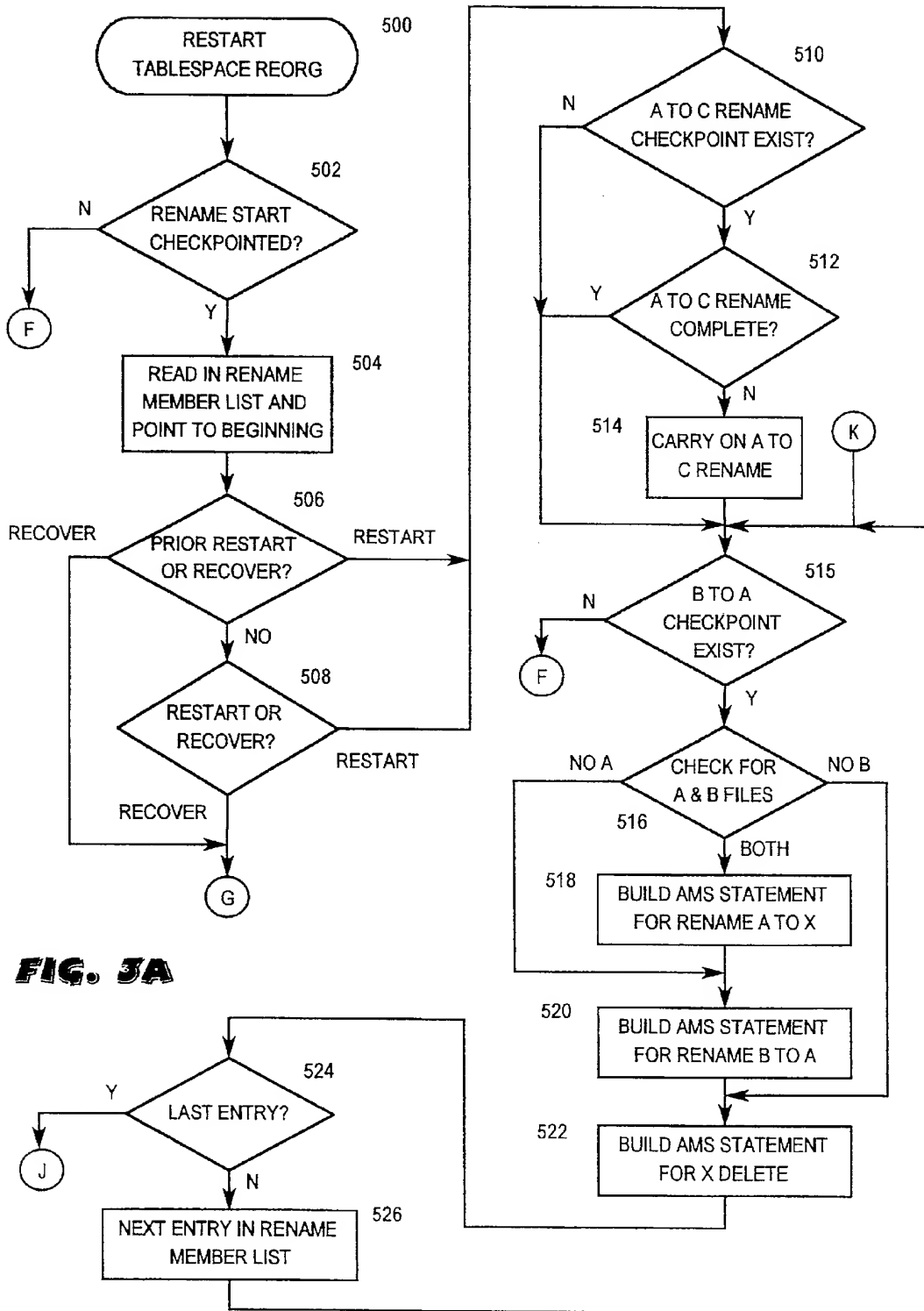
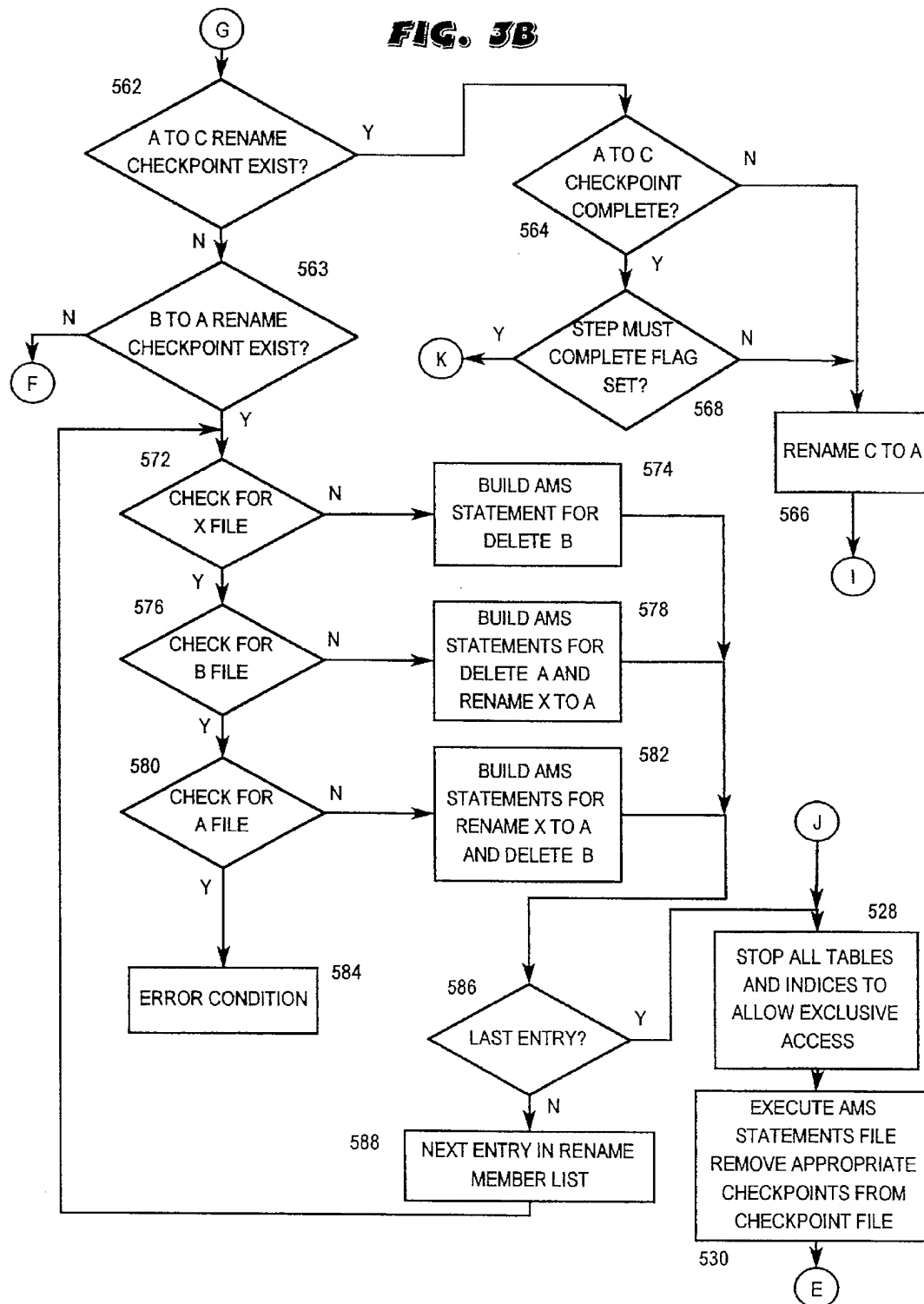
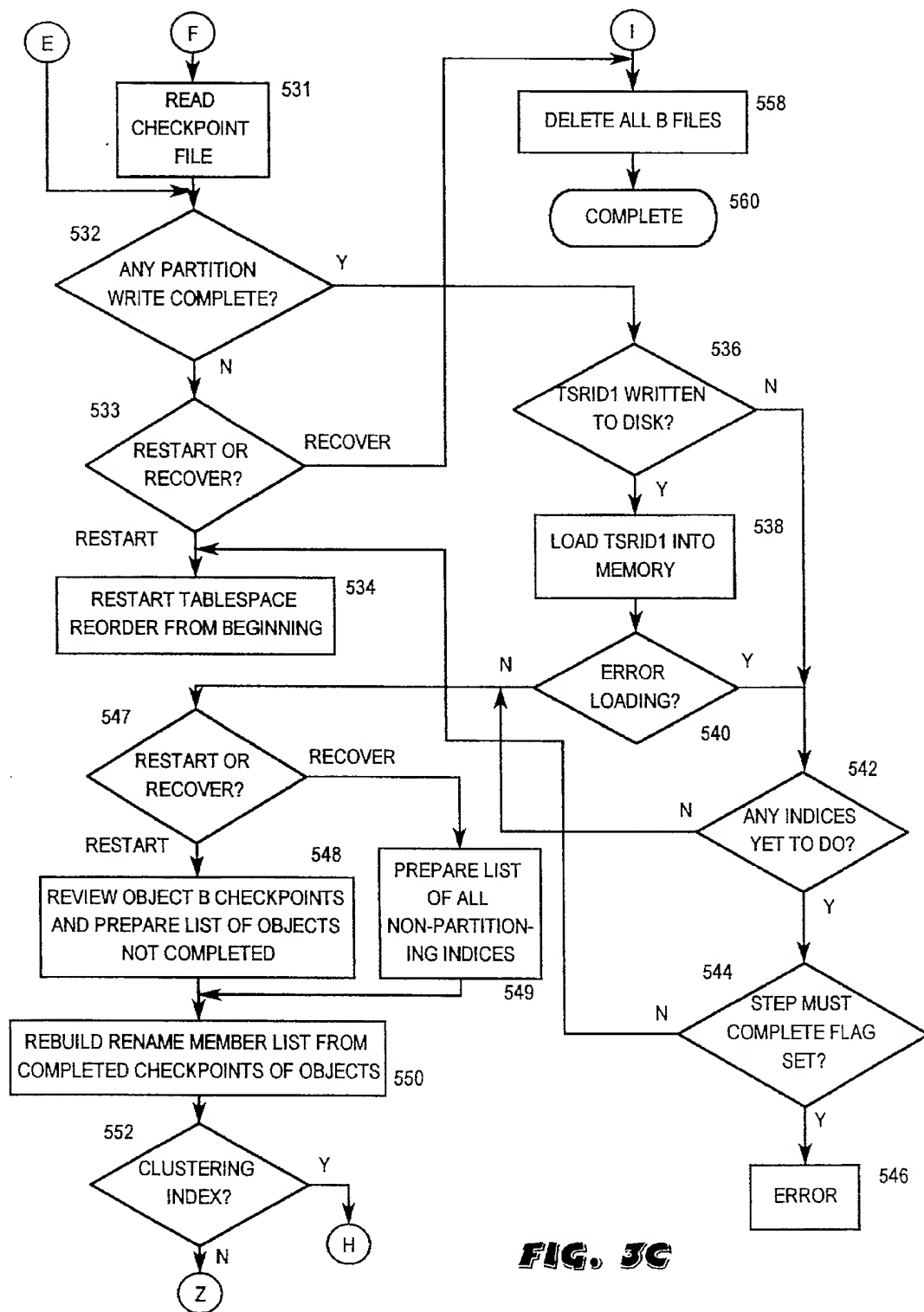


FIG. 3B

**FIG. 3C**

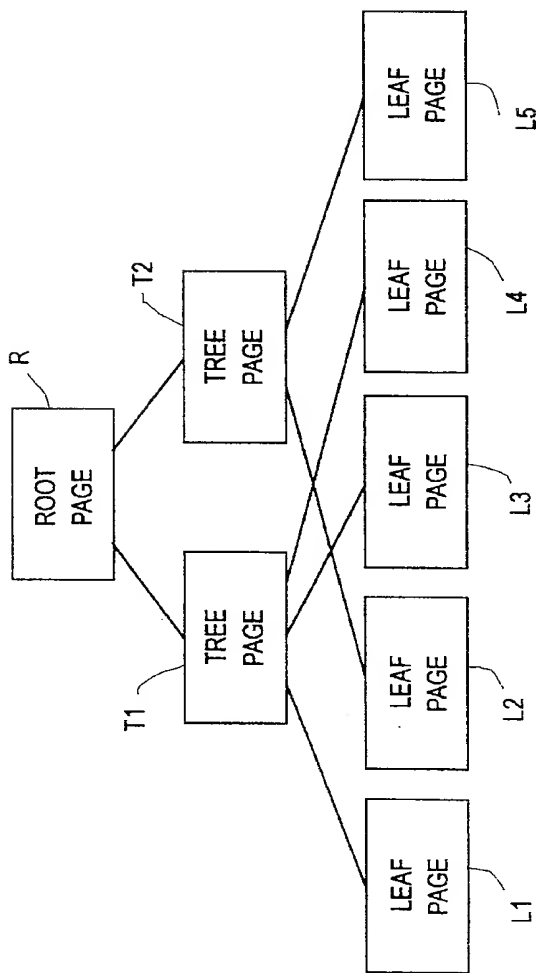
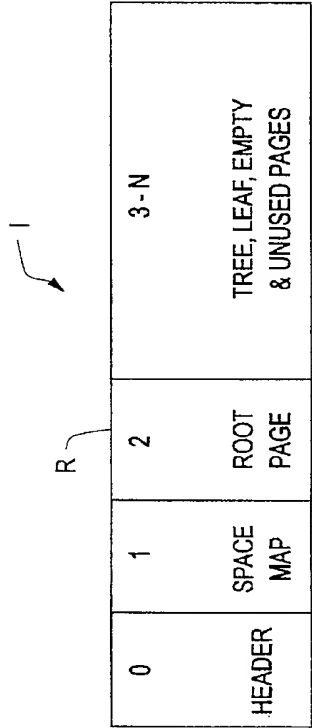


FIG. 4



VSAM LDS/ESDS FILE

FIG. 5

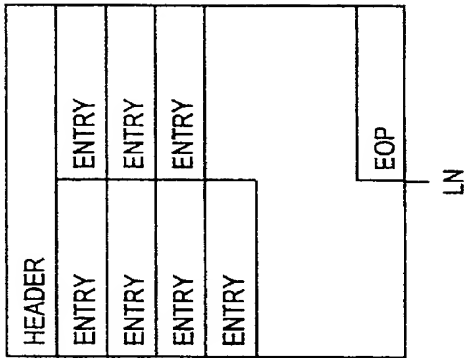


FIG. 7B

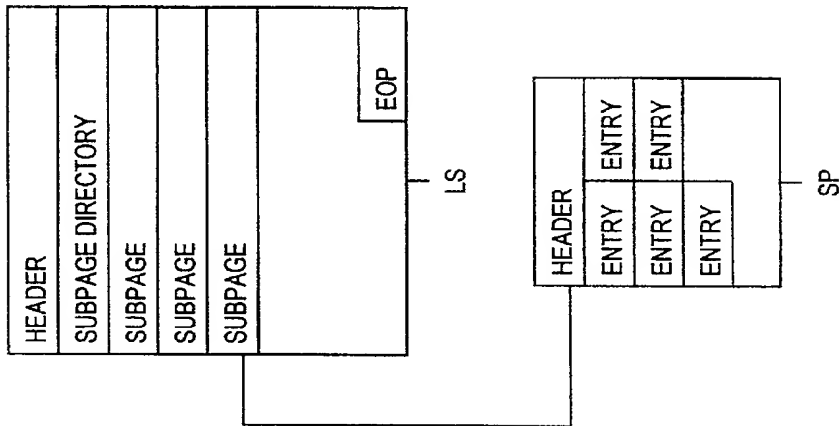
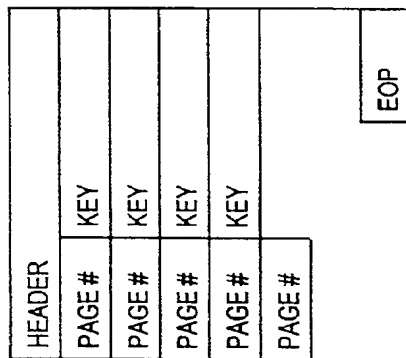


FIG. 7A



ROOT OR
PAGE TREE

FIG. 6

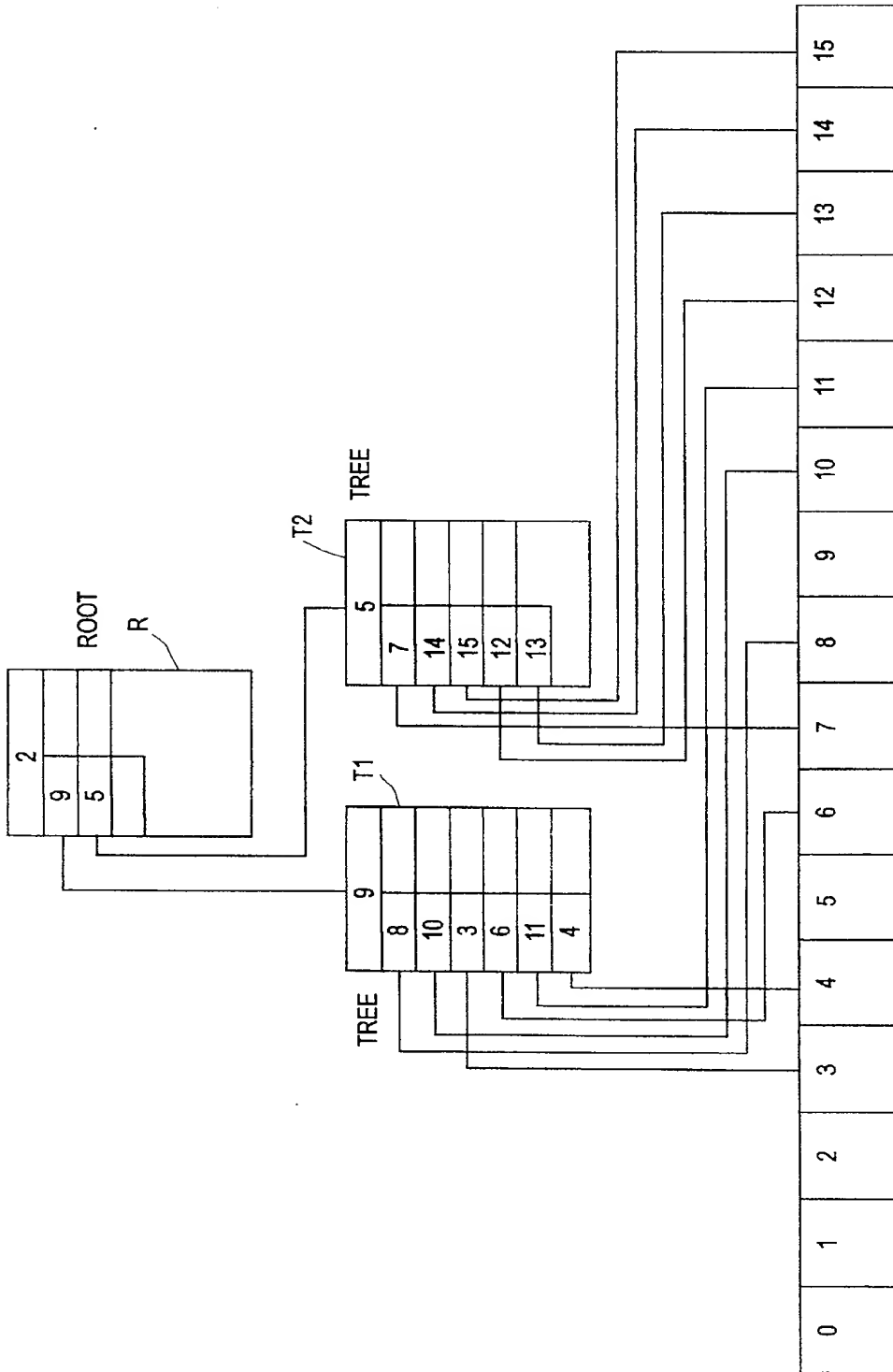


FIG. 8

LOGICAL	1	2	3	4	5	6	7	8	9	10	11
PHYSICAL	8	10	3	6	11	4	7	14	15	12	13

LB

FIG. 9

PHYSICAL	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
LOGICAL	0	0	0	3	6	0	4	7	1	0	2	5	10	11	8	9

PB

FIG. 10

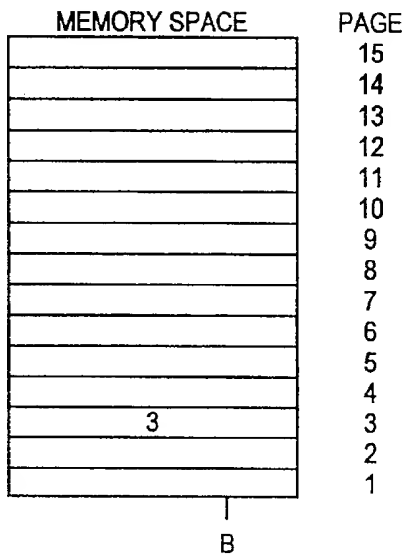


FIG. 11A

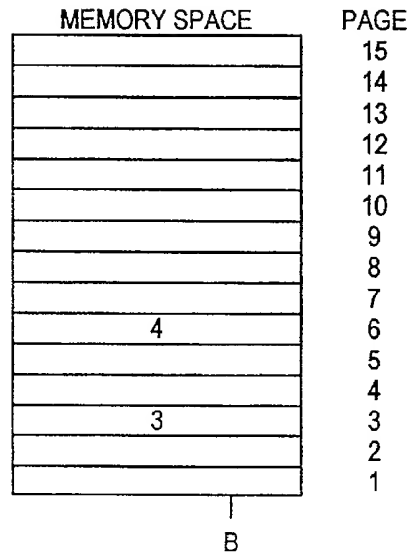


FIG. 11B

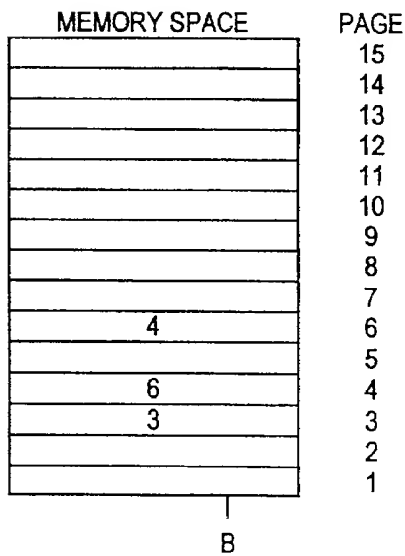


FIG. 11C

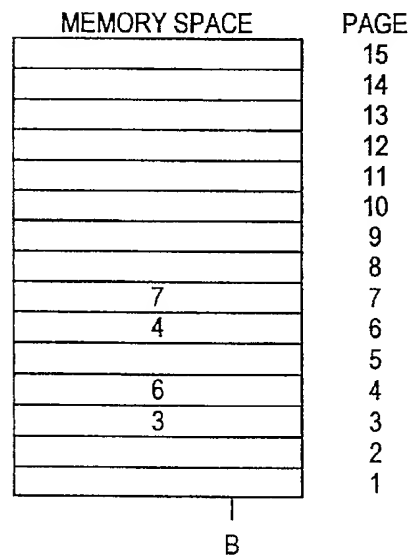


FIG. 11D

MEMORY SPACE	PAGE
	15
	14
	13
	12
	11
	10
	9
	8
7	7
4	6
	5
6	4
3	3
	2
8	1

B

FIG. 11E

MEMORY SPACE	PAGE
	15
	14
	13
	12
	11
	10
	9
	8
7	7
4	6
	5
6	4
3	3
10	2
8	1

B

FIG. 11F

MEMORY SPACE	PAGE
	15
	14
	13
	12
	11
	10
	9
	8
7	7
4	6
11	5
6	4
3	3
10	2
8	1

B

FIG. 11G

MEMORY SPACE	PAGE
	15
	14
	13
	12
	11
12	10
	9
	8
7	7
4	6
11	5
6	4
3	3
10	2
8	1

B

FIG. 11H

MEMORY SPACE	PAGE
	15
	14
	13
	12
13	11
12	10
	9
	8
7	7
4	6
11	5
6	4
3	3
10	2
8	1

B

FIG. 11I

MEMORY SPACE	PAGE
	15
	14
	13
	12
13	11
12	10
	9
14	8
7	7
4	6
11	5
6	4
3	3
10	2
8	1

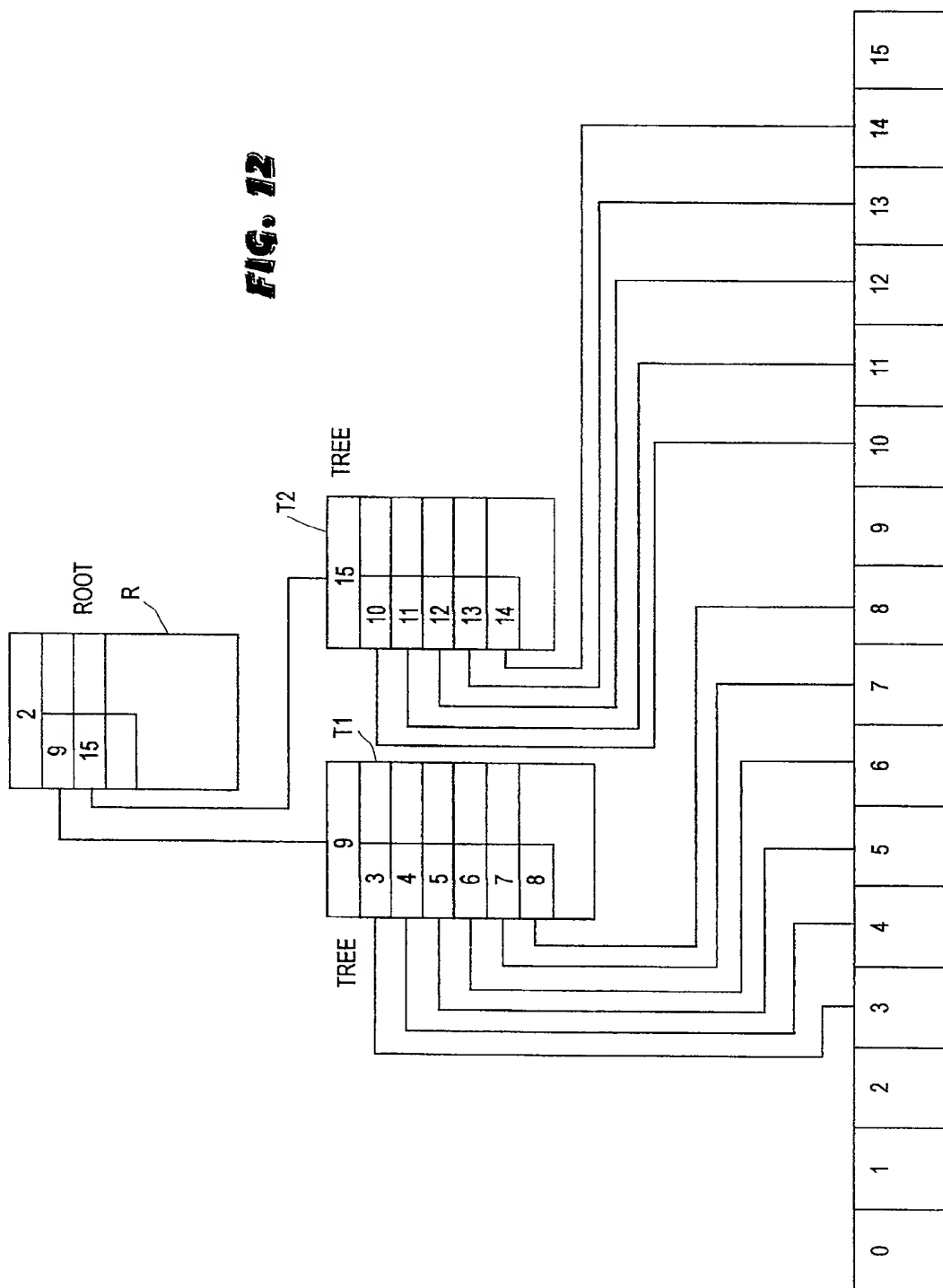
B

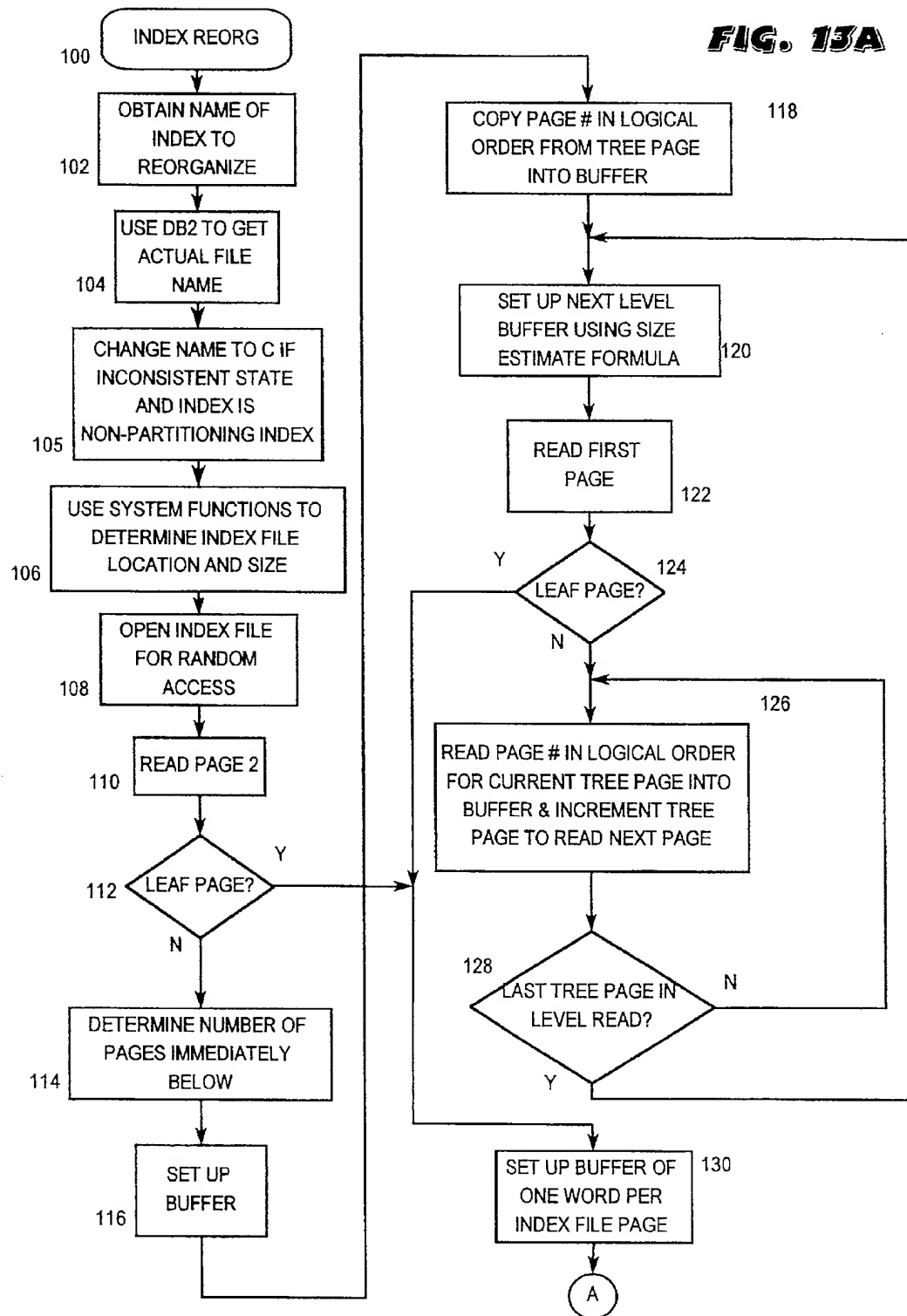
FIG. 11J

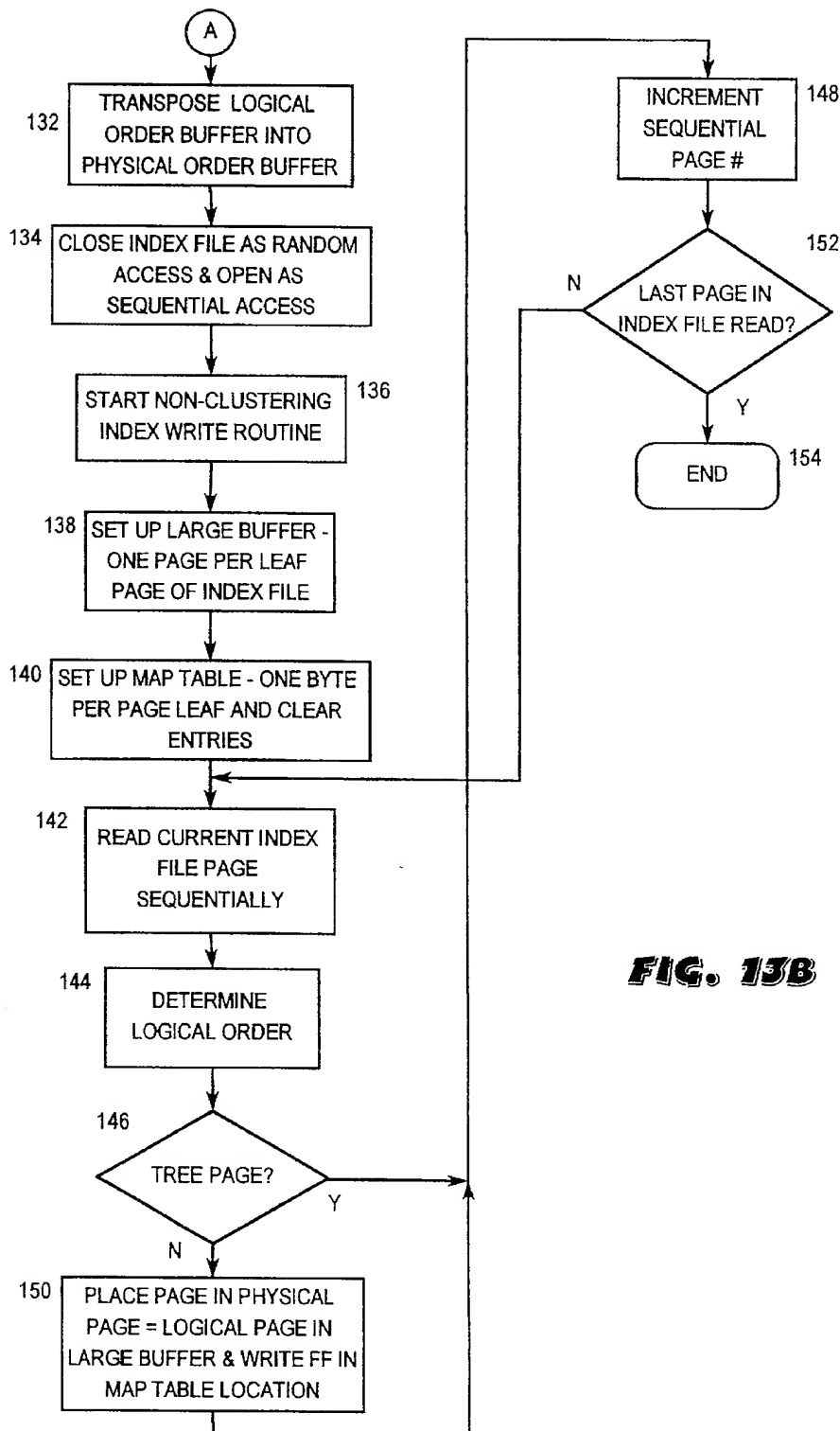
MEMORY SPACE	PAGE
	15
	14
	13
	12
13	11
12	10
15	9
14	8
7	7
4	6
11	5
6	4
3	3
10	2
8	1

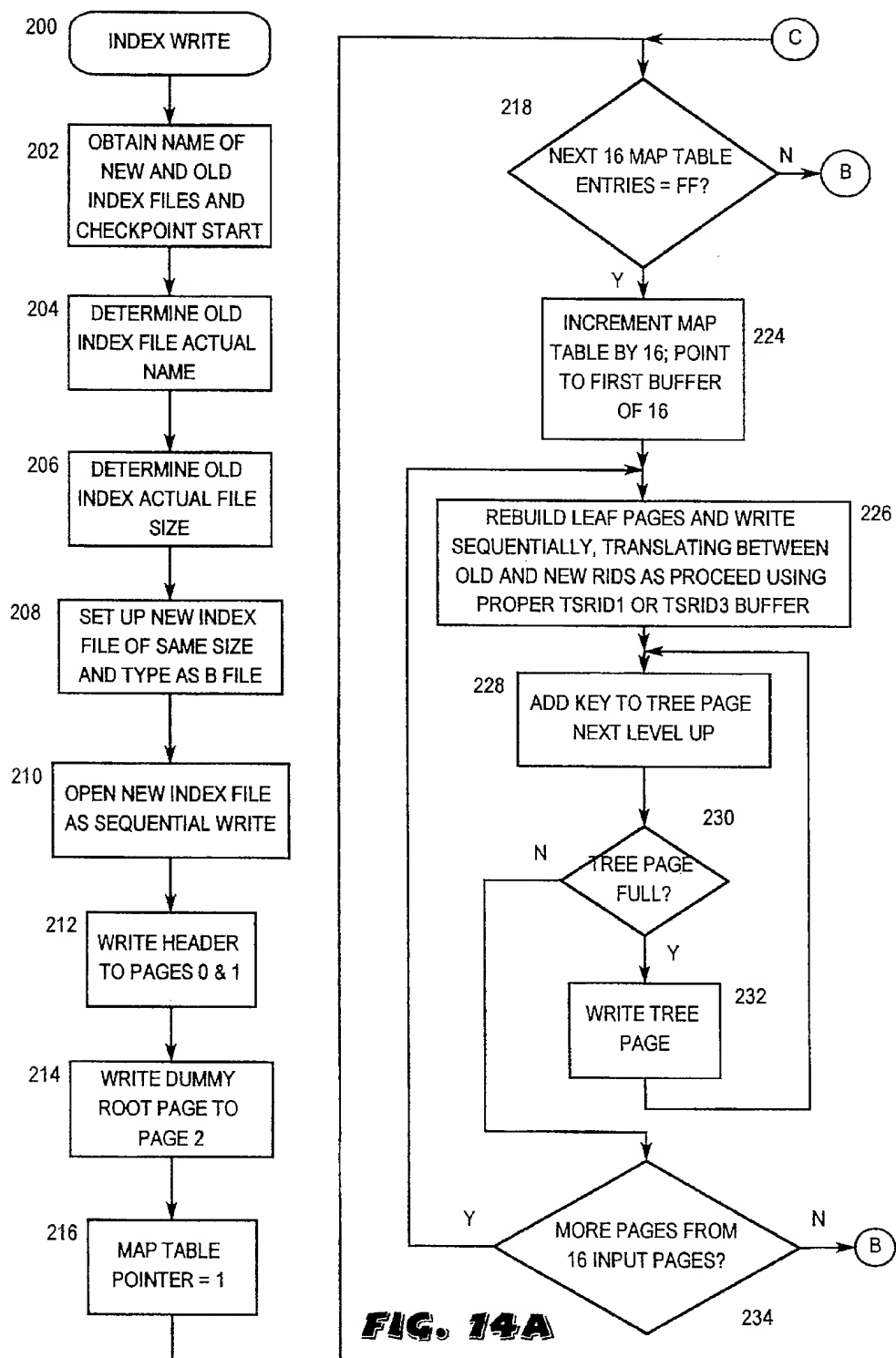
B

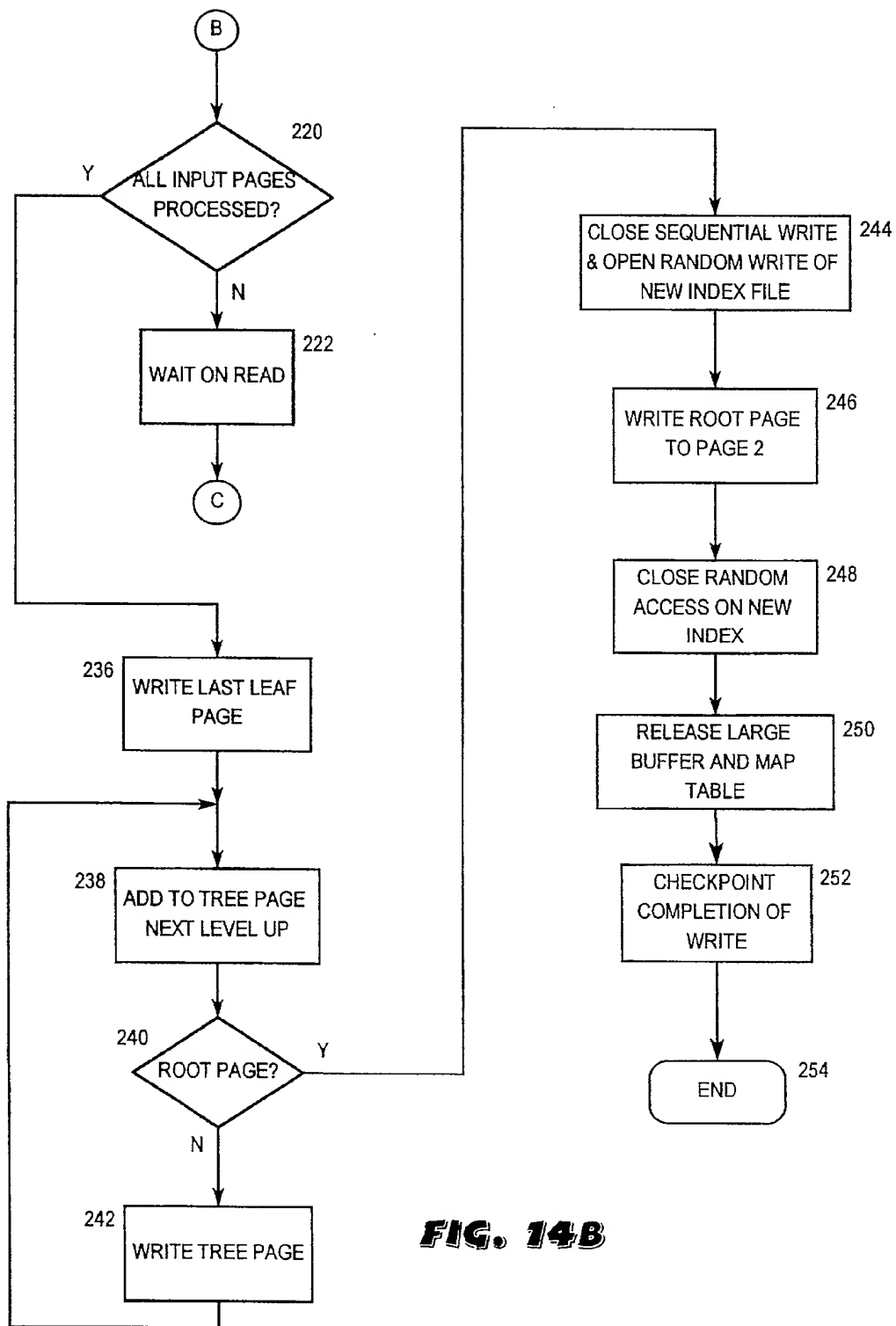
FIG. 11K





**FIG. 13B**



**FIG. 14B**

PAGE	0	1	2	3	4	5
ROW 01	HEADER	SPACE MAP	A		C	D
ROW 02					E	B

FIG. 15A

PAGE	0	1	2	3	4
ROW 01	HEADER	SPACE MAP	A	C	D
ROW 02			B		E

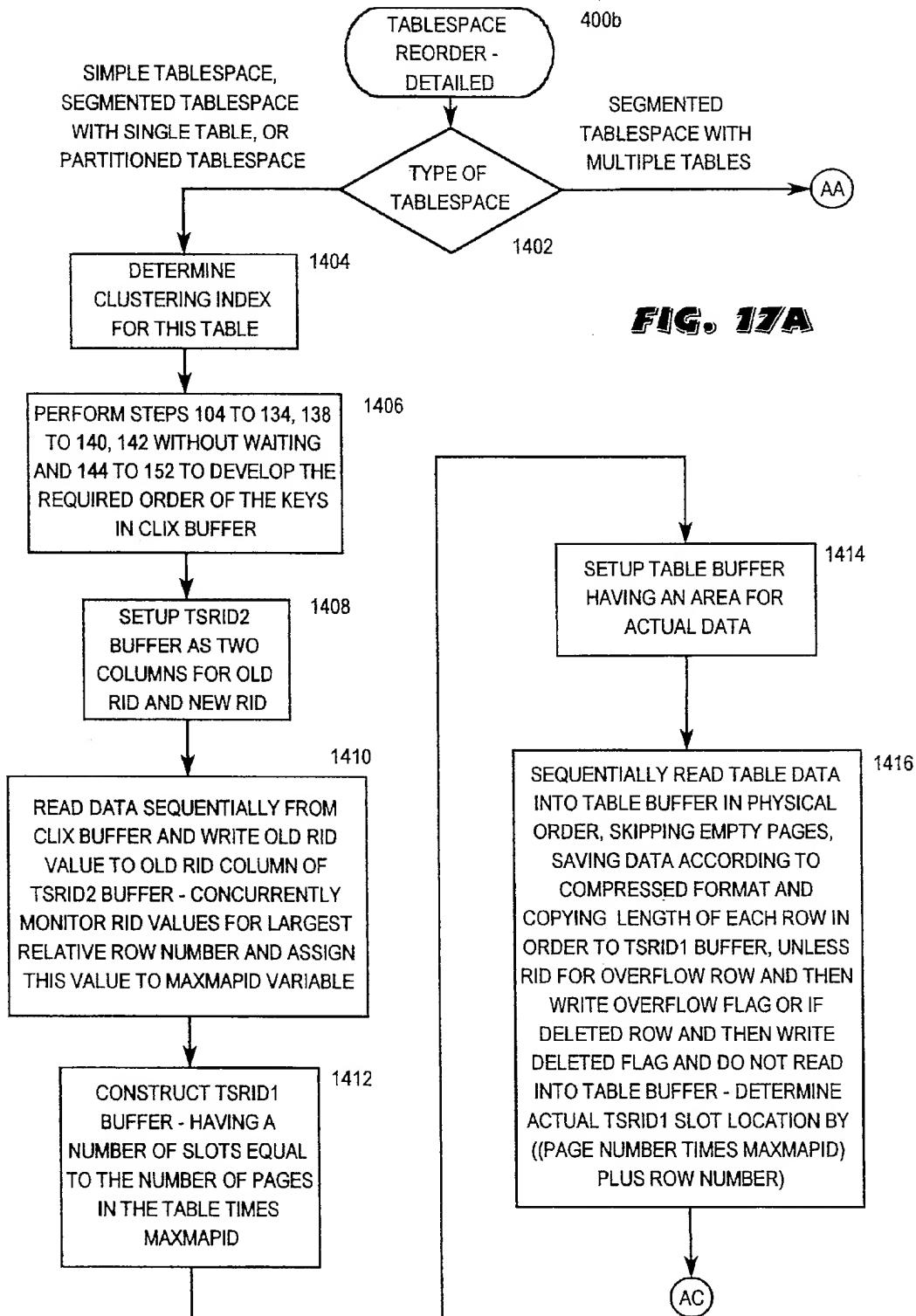
FIG. 15B

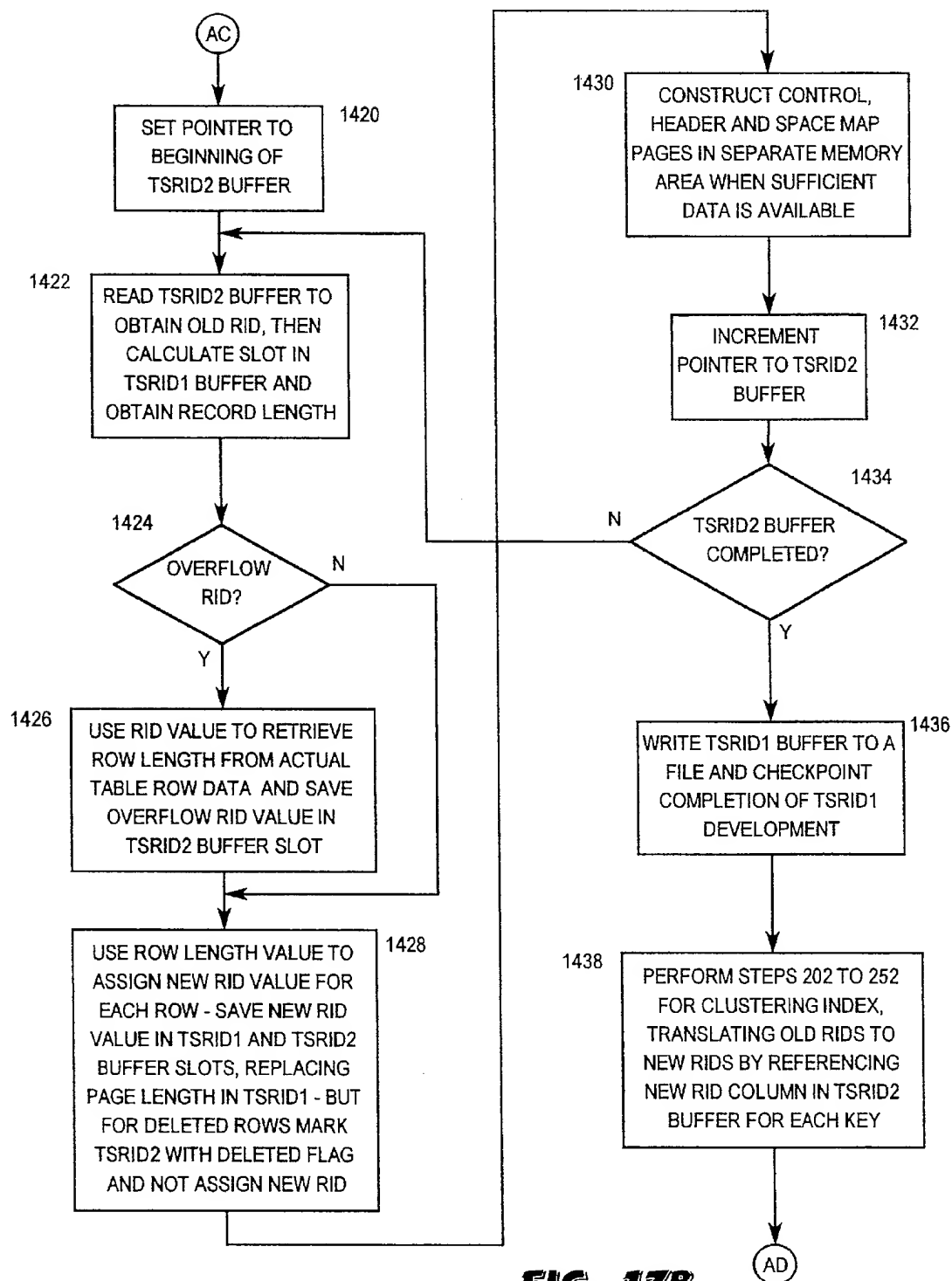
KEY	RID
A	201
B	502
C	401
D	501
E	402

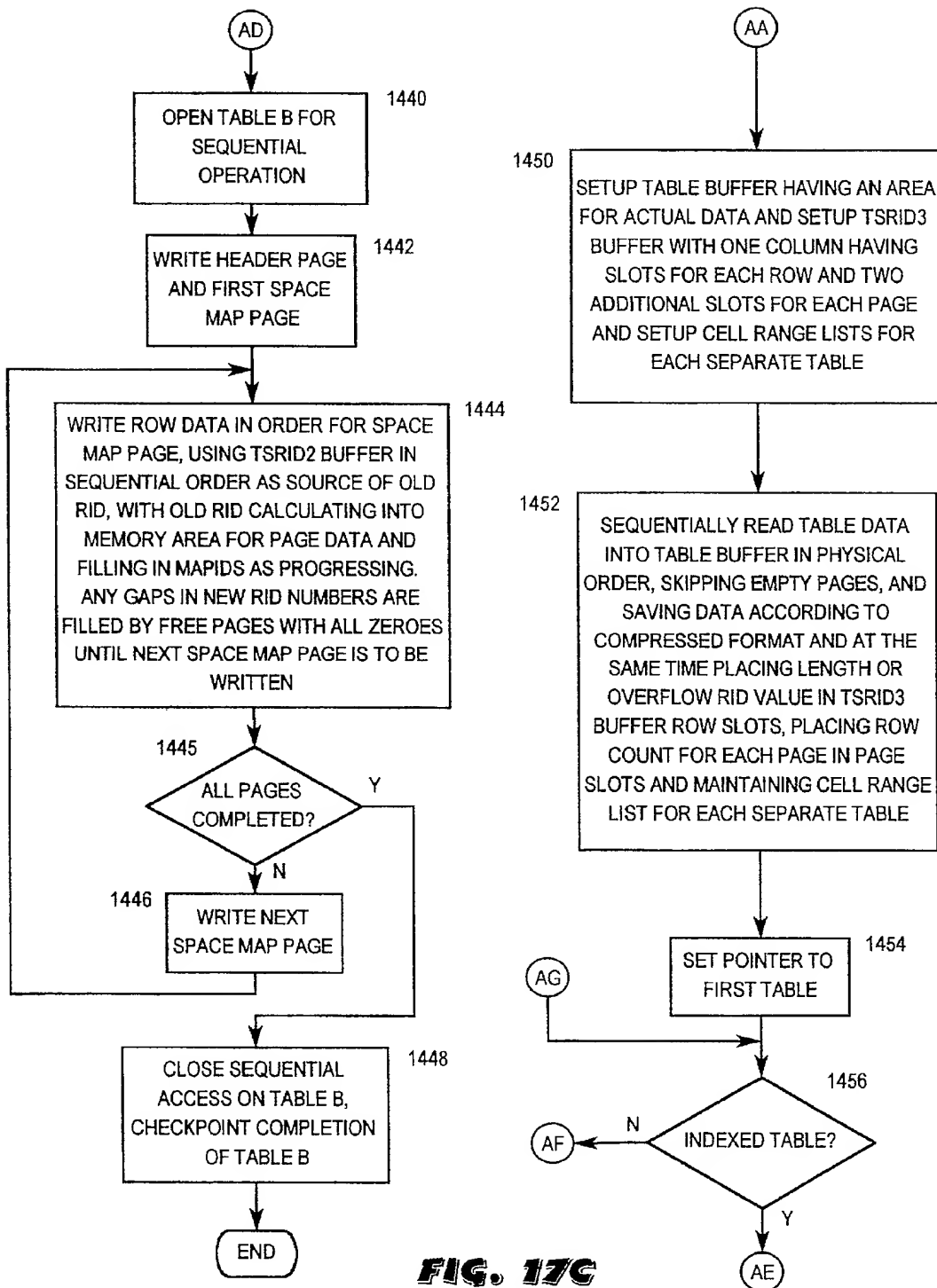
FIG. 16A

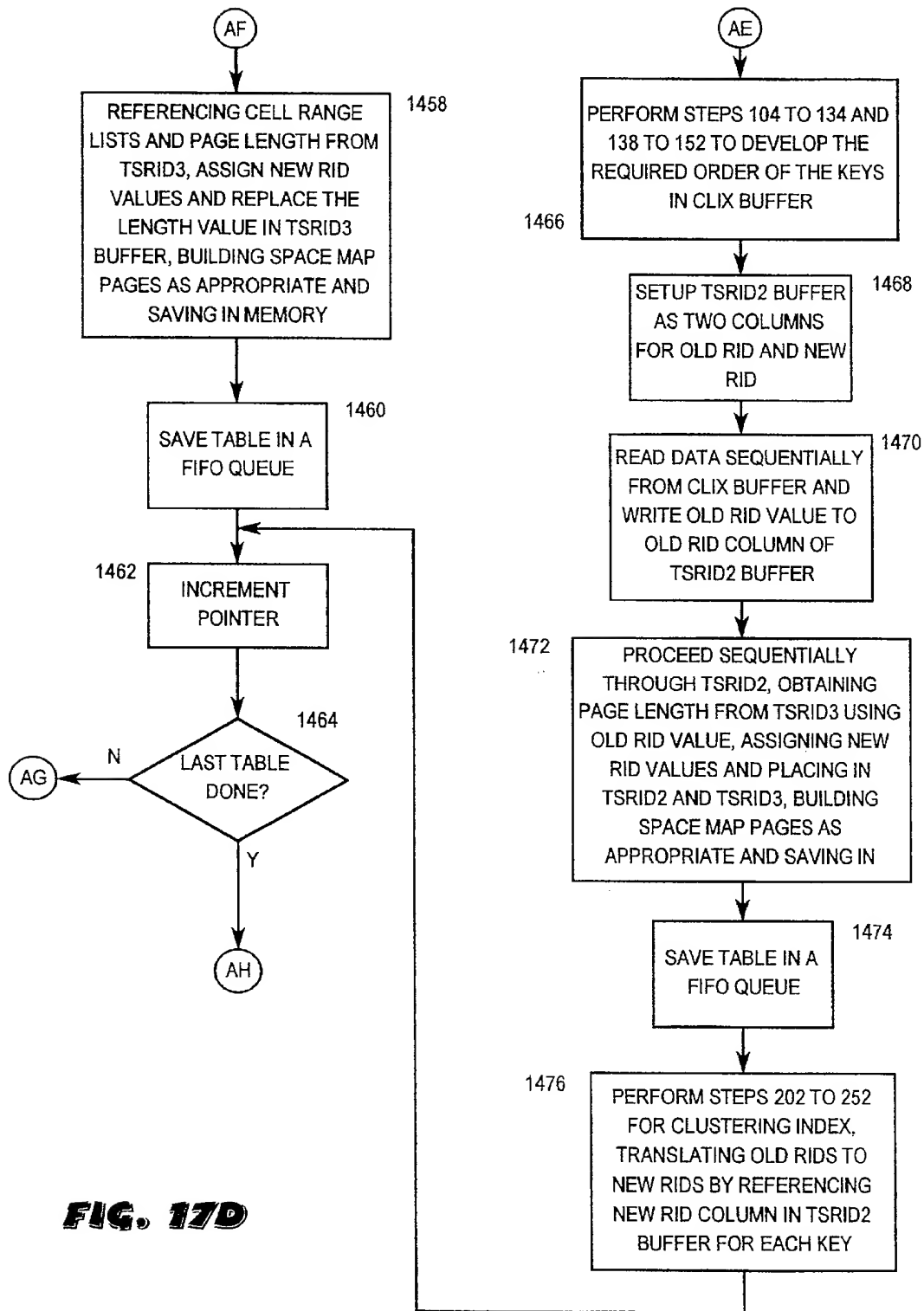
KEY	RID
A	201
B	202
C	301
D	401
E	402

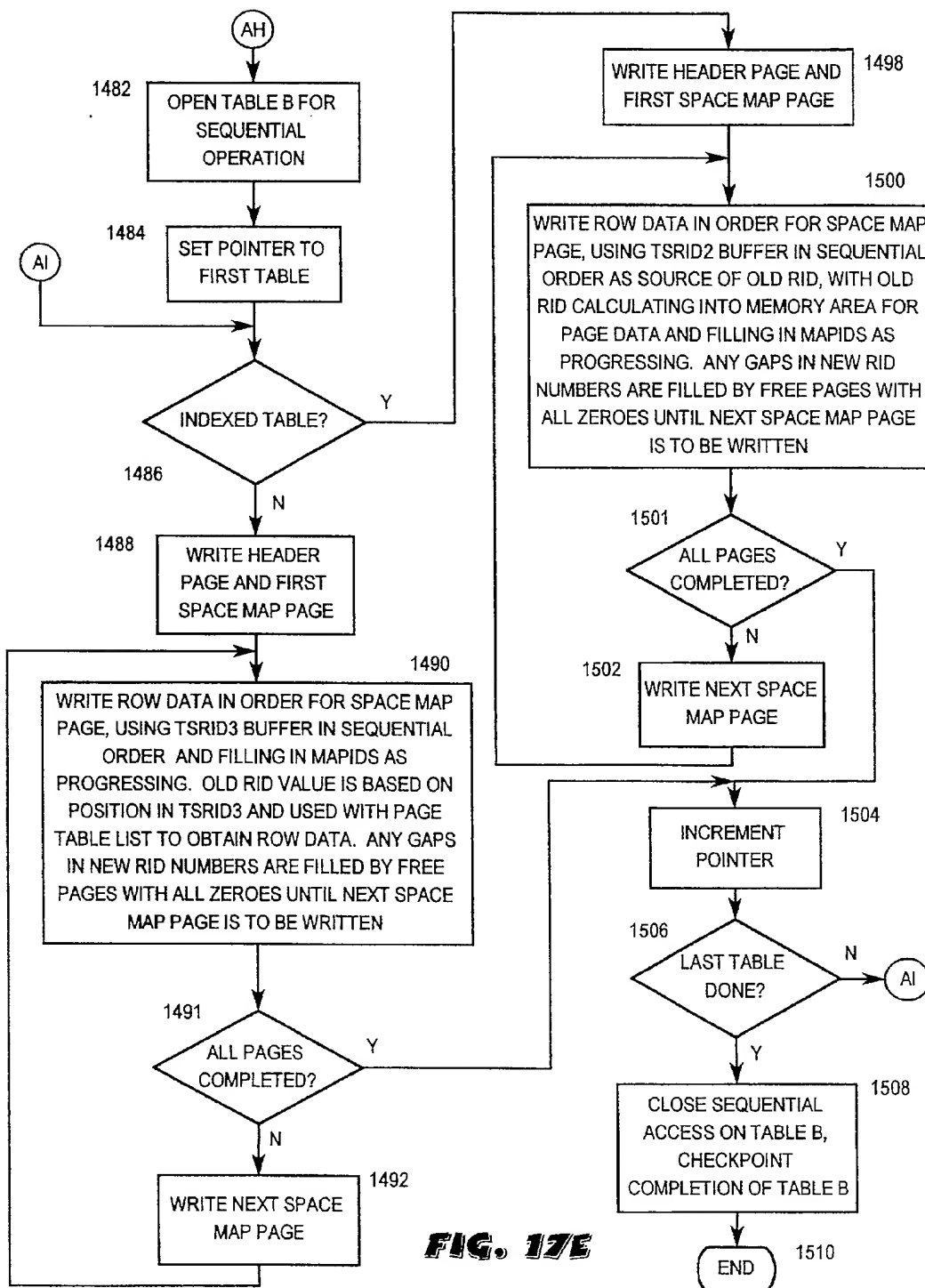
FIG. 16B

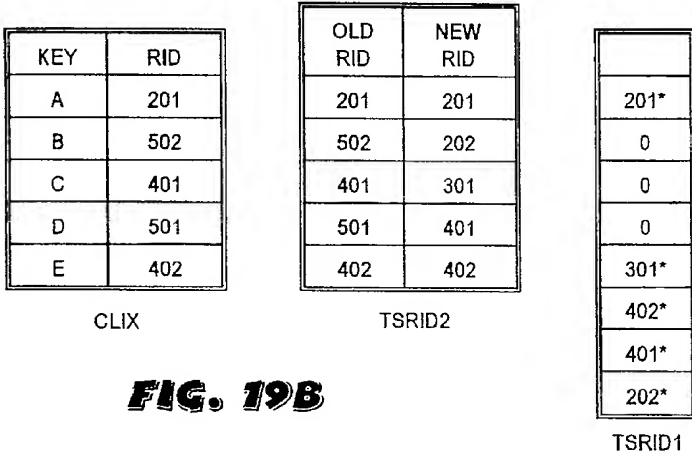
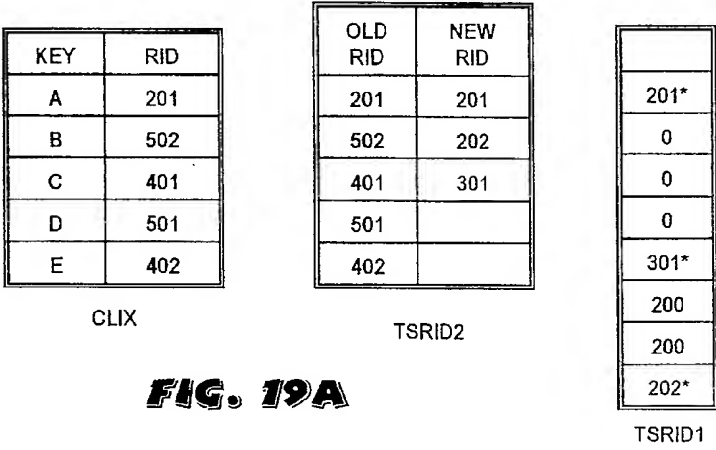
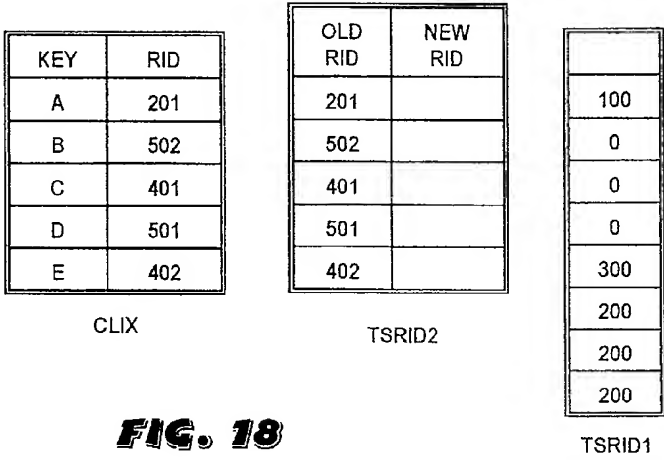


**FIG. 17B**



**FIG. 17D**

**FIG. 17E**



OLD RID	LENGTH
201	100
401	300
402	200
501	200
502	200

ILLUS. ONLY

TSRID3

FIG. 20A

OLD RID	LENGTH
201	201*
401	202*
402	301*
501	200
502	200

ILLUS. ONLY

TSRID3

FIG. 20B

KEY	RID	OLD RID	NEW RID	LENGTH
A	201	201		100
B	502	502		300
C	401	401		200
D	501	501		200
E	402	402		200

CLIX

TSRID2

TSRID3

FIG. 21A

KEY	RID	OLD RID	NEW RID	LENGTH
A	201	201	201	201*
B	502	502	202	301*
C	401	401	301	200
D	501	501		200
E	402	402		202*

CLIX TSRID2 TSRID3

FIG. 21B

KEY	RID	OLD RID	NEW RID	LENGTH
A	201	201	201	201*
B	502	502	202	301*
C	401	401	301	402*
D	501	501	401	401*
E	402	402	402	202*

CLIX TSRID2 TSRID3

FIG. 21C

**RESTARTABLE METHOD TO REORGANIZE
DB2 TABLESPACE RECORDS BY
DETERMINING NEW PHYSICAL POSITIONS
FOR THE RECORDS PRIOR TO MOVING
USING A NON SORTING TECHNIC**

SPECIFICATION

This is a continuation-in-part of application Ser. No. 07/889,454, filed May 27, 1992 now U.S. Pat. No. 5,408, 654.

BACKGROUND OF THE INVENTION

1. Field of the Invention

The invention relates to reorganizing database data and index files, particularly DB2 tablespaces, into key order without utilizing conventional sorting procedures, while allowing the tablespaces to be viewed during reorganization and allowing prompt recovery or restarting of the process if interrupted before completion.

2. Description of the Related Art

Databases are used on computers for a myriad of reasons. In many cases the databases are extremely large, having entries in the millions. When databases reach this size, and the information is needed in a transactional or real time basis, mainframe computers are utilized. International Business Machines Corp. (IBM) has developed a database environment referred to as DB2 for use on its mainframes. Given IBM's high market percentage in mainframes, DB2 is clearly the leading database system.

A tablespace is the DB2 term used to identify a database. Tablespaces can be simple, segmented or partitioned. In a simple tablespace, the data is kept in one file and there may be a clustering index and other indices. A clustering index is the index where the keys are kept in sequential order and the data is preferably kept in this same order. In a segmented tablespace, many different logical data files or tables are kept in a single file and there may be a clustering index and other indices for each logical data file. There are no indices directed to multiple logical files. In a partitioned tablespace, the data is kept in different files, but there is a clustering index for each file. There may be additional indices directed to all of the partitions.

DB2 uses a balanced tree index structure. In this structure, root, tree and leaf pages are used, with each page at each level containing the same number of entries, except of course the last one. The leaf pages are the lowest level and each contains a number of entries referring to the actual data records contained in the DB2 data tables. Each leaf page is maintained in internal logical order automatically by DB2. Tree pages are the next level up, and are used to indicate the logical order of the leaf pages. For large databases, there may be many several layers of tree pages, a higher level of tree pages referencing a lower level of tree pages. Ultimately the number of tree pages is reduced such that all the entries or references fit into a single page referred to as the root page. As in leaf pages, within each tree or root page the entries are kept in logical order by DB2.

The data tables or files are organized into pages. The first page of each table is a header page. The second page and certain pages thereafter are referred to as space map pages. The header page contains information relating to the entire table while the space map pages include information relevant to free space in a following number of data pages. The actual frequency of the space map pages is based on

tablespace characteristics and is well known. The remaining pages are data pages. Up to 255 rows or records of data may be present in a single page. The actual number of rows depends on the size of the row and the size of the page. Each row receives a unique identifier, referred to as the RID, which identifies the page number and relative row number in that page.

One problem with any database is that the physical location of the various pages often becomes quite scattered. This is true for the data pages in the tables and the leaf pages in the indices. A disorganization also develops between the clustering index and the data, so that the table data is no longer physically in its intended logical order. This scattering results in reduced performance as now the storage device must move between widely scattered physical locations if logically sequential operations are to be performed. This is true of whatever type of Direct Access Storage Device (DASD) is used to store the file. Therefore the files need to be reorganized periodically so that the logical and physical ordering of the pages better correspond, thereby improving performance of operations.

IBM provides utilities with DB2 to reorganize the entire tablespace and just the index files. Several other third-party DB2 utility providers also have tablespace and index reorganization packages. These packages usually operate in the same manner. First, the entire file, either tablespace or index, is read in physical order. Each page in the file is then separated into its component record entries. Next, the record entries are sorted by key value using a conventional sort package. Finally, the sorted records are rewritten back into the file. While this process may sound simple, it must be understood that quite often there are hundreds of thousands to millions of entries in the file. When this number of entries is considered, then the process becomes quite time consuming, particularly the sorting step. The third party packages are faster than IBM's utility, but primarily because the sort packages used are more efficient and also because they use standard available sort package facilities, such as sort exits to reduce intermediate file I/O. So even in those cases the process is quite tedious and is done less frequently than desirable, so overall database performance suffers. Therefore it is desirable to have a significantly faster DB2 table and index reorganization method, so that the tables and indices can be reorganized more frequently and overall operations made more efficient.

Additionally, because the reorganization procedures often take large amounts of time, it would be desirable to access the files for viewing purposes, if not for updating purposes, during the reorganization. Further, should for some reason the process be stopped before completion, it would be desirable to have several alternatives to become fully operational without redoing the entire process.

SUMMARY OF THE INVENTION

The present invention relates to an improved method to dramatically reduce the time required to reorganize tablespaces and index files, particularly those as used by DB2. Additionally, the operations allow viewing access during the reorganization process; are non-destructive, allowing a prompt return to the original state; and are checkpointed, allowing restarting at selected intervals to avoid having to completely restart the process.

Briefly, the process is started by flushing any pending operations and setting all of the files in the tablespace to read only status. By using read only status, viewing access is

allowed during the reorganization process. The first clustering index of the tablespace is then determined and the tablespace reordering and clustering index reorganization sequence is started. The original table and indices are considered as A files and read into memory. New row IDs or RIDs are developed as described below so that the order of the data is developed. After the new RIDs have been developed, both the clustering index and the row data are read out of memory and written to a new table and clustering index files in the proper order as B files. After the completion of writing the B files, checkpoints are placed to indicate sequencing of events.

If the tablespace is partitioned and there are any certain non-clustering, non-partitioned indices, then a series of AMS statements are developed and executed which rename all of these A or original files to C or temporary files as soon as the first reorganization of any single partition is completed. This is done as the indices will now be unusable because the references will no longer be correct. These AMS statements are then executed and checkpointed. Next, all files which need to be renamed are determined and written to a list. All of the table files are then stopped to allow exclusive access. Next, a series of AMS statements are built to do the renaming operations. Specifically, a series of statements are built to rename all of the A or original files to X or old files. Then a series of statements are developed renaming all B or new files to A files. Finally, a series of statements are made deleting all of the X files, thus effectively completing the task. After these statements have been developed they are executed and after execution, all checkpoints for the partition are removed. If there are any more partitions remaining, then control returns to the tablespace reordering and clustering index reorganization sequence to proceed with the next partition. This loop continues until all partitions have been processed. Thus, by use of the original and new files, the original can remain in a read only state until the new file process is complete. This allows viewing during almost the entire reorganization process and the process can be returned to the original state if possible or can be restarted from approximately where it was halted.

If the tablespace is not partitioned or after all partitions have had their tables reordered and clustering indices reorganized, then any remaining non-clustering indices are reorganized. This sequence is described in detail in the detailed description. Simply, again the original files are read into memory and the new files as they are written out are written into B files. In this manner the original A or C files are always remaining until after the development of the new files. After the new files have been developed, the files are renamed as described above.

The restarting process reviews the various check points and presence of the various A, C, X or B files and based on the checkpoint status and the presence of the various files, properly proceeds from the interrupted portion onward.

The tables and clustering indices are reordered in a different manner than the non-clustering indices. The non-clustering indices will be described first.

As a first general operation, the logical order of the leaf pages is determined by accessing the index tree. A buffer is used to receive the physical page number of the leaf pages in logical order. This buffer is then transposed into a physical order buffer, with the logical order value associated with the physical order value. After this, a large buffer is set aside in memory to receive a copy of all the leaf pages in the index file. The index file is then read in a sequential operation. As each physical page is read, the transposed or physical order

buffer is accessed to find the proper logical page. If a leaf page has just been read, it is then placed in the large buffer in its logical position. When the entire index file is read and the leaf pages written to the large buffer, the large buffer is in a reorganized logical order. These leaf pages are then sequentially written to the new or B index file, referencing a buffer containing a translation between the old and new RID values so that the new RID values are written into the index, with the tree pages interspersed as they are filled. When the write operations are completed, the B or new index file replaces the A or old index file as described above.

Preferably the writing of the B index file can occur concurrently with the filling of the large buffer. As the beginning of the buffer becomes filled, the beginning entries can be written to the new index file, so that the write process is effectively racing the read and fill processes. If the index is not badly scattered, the concurrence can further reduce the total required time of the reorganization process.

As noted, table and clustering index reorganization proceeds somewhat differently. If a simple tablespace, a segmented tablespace having just a single table or a partitioned tablespace, then the clustering index for the particular table or partition is first reorganized basically as discussed above. Then a second or TSRID2 buffer is set up in order of the new index that has columns for the old RID and the new RID values. As the data is read sequentially from clustering index buffer, the old RID value is written to the old RID column of the TSRID2 buffer and the row numbers are monitored to determine the maximum row number. Then another buffer, the TSRID1 buffer, is developed having a number of slots equal to the number of pages times the maximum row number. Finally, a table buffer is set up having an area to receive the actual data in a compressed format. The table data is then read into this table buffer in sequential order, skipping any empty pages and saving the data. The length for each row is transferred to the proper slot in the TSRID1 buffer. A pointer is then placed at the beginning of the TSRID2 buffer, which is then read sequentially to obtain the old RID value which is used to calculate a slot in the TSRID1 buffer, and the row length is obtained. If an overflow RID is indicated, the actual row length is obtained using the overflow RID value. With the row length obtained, new RID values can be sequentially assigned from the beginning of the table and placed in both the TSRID1 and TSRID2 buffers. Header and space map pages are developed as necessary in a separate area in the table buffer as the necessary information is available. This process is continued through the entire TSRID2 buffer. The TSRID1 data space is then written to a file and checkpointed for recovery or restart reasons. The clustering index as reorganized is then written to a B file, effectively using the writing steps defined above. Then the table B file is opened for sequential operations and the header page and first space map page are written. Then using the TSRID2 data space in sequential order as an ordering element, the old RID value is used to reference the actual page data from the table buffer, which is then written in this order, along with the new RID numbers as stored in the TSRID2 buffer. The space map pages are written as necessary. After this process has been completed, sequential access to the table B file is closed and completion of the table B write operation is indicated.

Segmented files proceed slightly differently, in that first a TSRID3 buffer having a column to receive the row length and two additional slots for each page is set up. The table buffer having areas for the row data and header and space map pages is also setup. Further, cell range lists are setup for each table. The table data is then read into the table buffer

in physical order. Additionally, the row length is placed in the TSRID3 buffer at the same time, with the row count for each page being placed in the page slots. Additionally, the cell range lists are maintained for each separate table in the segmented tablespace. If there is no clustering index, reference is simply made to the cell range lists and the page lengths from the TSRID3 buffer, new RID values are assigned and replace the length value in the TSRID3 buffer, with space map pages being built as appropriate. The table data is then saved in a FIFO queue. This process is repeated for each table.

If the table has a clustering index, then the index is then reorganized as above, the TSRID2 buffer being filled to obtain the old and new RID values. Proceeding sequentially through the TSRID2 buffer, the page length is obtained from the TSRID3 data space, with new RID values then being assigned and placed in the TSRID2 buffer, with space map pages being developed as appropriate. Again the table is saved in a FIFO queue for writing. Finally, after this step is completed, the clustering index is written out, with translation of the RIDs occurring as described above. This process also repeats until all tables have been completed.

Once the last table is completed, the table B file is opened for sequential operation and the header and first space map pages are written. Then the row data is written out in sequential order using either the TSRID3 buffer, if it is not an indexed table, or using the TSRID2 buffer to determine the old RID and new RID numbers. After all the data values have been retrieved from memory and written to the table B files for all tables, sequential access is closed and a checkpoint is completed.

Therefore the reorganization is also performed without a conventional sorting process, greatly speeding up operations.

BRIEF DESCRIPTION OF THE DRAWINGS

A better understanding of the present invention can be obtained when the following detailed description of the preferred embodiment is considered in conjunction with the following drawings, in which:

FIG. 1A and 1B are a flowchart of a DB2 tablespace reorganization procedure according to the present invention;

FIG. 2 is a flowchart of the tablespace reorder sequence simplified of FIG. 1A;

FIGS. 3A, 3B and 3C are a flowchart of a restart sequence according to the present invention;

FIG. 4 is a representation of a balanced tree structure;

FIG. 5 is a representation of the page order of a DB2 index file;

FIG. 6 is a representation of the structure of a root or tree page in a DB2 index file;

FIGS. 7A and 7B are representations of the structures of the leaf pages in a DB2 index file;

FIG. 8 illustrates the linkages in an exemplary DB2 index file tree structure;

FIG. 9 illustrates the organization of an exemplary buffer organized in logical order of leaf pages;

FIG. 10 illustrates the organization of the buffer of FIG. 9 in physical order;

FIGS. 11A-11K illustrate the filling of a large memory space buffer in logical order with leaf pages obtained in physical order;

FIG. 12 illustrates the linkages of the final DB2 index file of FIG. 8 after reorganization;

FIGS. 13A, 13B, 14A and 14B are flowcharts of sequences for reorganizing DB2 non-clustering indices;

FIGS. 15A and 15B are representations of a DB2 table before and after reorganization;

FIGS. 16A and 16B are representations of a clustering index buffer before and after reorganization;

FIGS. 17A, 17B, 17C, 17D and 17E are a detailed flowchart of a sequence for reorganizing DB2 tables and clustering indices;

FIG. 18 is a representation of various buffers used with the sequence of FIGS. 17A-C;

FIGS. 19A and 19B are representations of the buffers of FIG. 18 during and after reorganization;

FIGS. 20A and 20B are representations of various buffers used with the sequence of FIGS. 17A and 17C-E during reorganization; and

FIGS. 21A, 21B and 21C are representations of various buffers used with the sequence of FIGS. 17A and 17C-E during and after reorganization.

DETAILED DESCRIPTION OF THE PREFERRED EMBODIMENT

Referring now to FIGS. 1A and 1B, a tablespace reorganization sequence 300 according to the present invention is illustrated. The tablespace reorganization sequence 300 commences at step 302, where all the various buffers in the computer relating to the tablespace are flushed so that there are no pending operations, particularly write operations, to the tablespace. Control proceeds to step 304, where all of the files in the original tablespace are set as read only. This read only status allows viewing access to the tablespace but does not allow updates as that would interfere with the reorganization process. However, by making the tablespace read only, at least viewing rights are provided so that complete blocking of the tablespace is not necessary and some operations can be performed. Control then proceeds to step 306, where the first clustering index of the tablespace is read. There may be tablespaces where no clustering indexes are used, but that is a very simple case and is not addressed in this description. Control proceeds to step 308, where a tablespace reorder and clustering index reorganization sequence 400a, 400b is commenced. This sequence will be described in summary shortly below as sequence 400a and in great detail further below as sequence 400b.

After the sequence 400a, 400b has been started, control proceeds to step 310 to determine if this is a partitioned tablespace. If so, control proceeds step 312 to determine if there are any non-clustering, multi-partition indices, that is, any indices directed not to a single partition but rather to all of the partitions. If so, control proceeds to step 314 to determine if this branch has been previously taken. If not, control proceeds to step 316, where a series of AMS statements for renaming the indices are built. Specifically, the indices are to be renamed from what are referred to as the A or original files to C or temporary files. This must be done at this time because after the first table reorder is commenced, the non-clustering, multi-partition indices will no longer be proper and therefore can not be utilized, even for just view access. After the AMS statements have been built, control proceeds to step 318 where a checkpoint is written to a checkpoint file to indicate that the A to C rename execution has started. An INCONSISTENT STATE flag is also set in the checkpoint file for later use. Control then proceeds to step 320, where the AMS statements are started.

Control proceeds from step 320 or from step 312 if there are no non-clustering, multi-partition indices or step 314 if the process has previously been performed, to step 326, where a list of files relating to the particular partition or table is written to a file referred to as the rename member list. This is the list of files relating to the particular table, such as the clustering index and other partition or segment indices, that need to be renamed as part of the process. Control proceeds to step 328 where a checkpoint is written to indicate starting of the renaming phase. Control proceeds to step 330, where all of the tables and indices are stopped so that exclusive access is granted for the renaming steps. This is necessary because when renaming is operating, even viewing access must be disabled. However, the steps are relative quick and therefore this is not considered to a major burden, with view access provided for the predominant portion of the entire process. Control proceeds to step 332, where a series of AMS statements are built. The first series of statements renames all of the A files, i.e. the original files, to X files. Then there are a series of AMS statements relating to conditional testing to determine if there were any failures during the renaming operation of A to X. If so, the tests cause aborting of the operation. The next series of statements that are built relate to renaming all of the B files, i.e. all of the newly created files developed by the reordering and reorganization sequences, to A files, i.e. to the names of the original files. Next is a series of conditional test statements to determine if there were any failures during the proceeding renaming operations and causing aborting if so. The final series of AMS statements built are those that delete of all X files.

Control then proceeds to step 334 where the AMS statement file is started. During operation of the process started at step 334 all of the original files will have been named to temporary X files, all of the new files will be named to the A files and then all of the temporary X files will be deleted. By the completion of the process, all of the original files will be completely replaced with all of the new, reordered sequential files. Control then proceeds to step 336, where the appropriate checkpoints are removed from the checkpoint file so that should operations have to be restarted, as described below, there is no indication that this process is not fully completed. Further, a flag called the STEP_MUST_COMPLETE flag is set in the checkpoint file, as once the process of step 334 has started, all of the operations must complete. Control then proceeds to step 322 to determine if the last partition has been finished. If not, then control proceeds to step 324, where the next clustering index for the tablespace is read. Control then proceeds to step 308 where the tablespace reorder and clustering index reorganization is started for that particular partition.

If this was not a partitioned tablespace as determined in step 310 or the last partition had been finished as determined in step 322, control proceeds to step 340. In step 340 the index reorganization sequence 100 is started for each of the remaining non-clustering indices, including any non-clustering, multi-partition indices. Control then proceeds to step 342 where the checkpoint file is read to determine the files that have to be renamed and these are placed in a rename member list for each index. Control proceeds to step 344, where starting of the renaming phase is written to the checkpoint file to allow later restarting. Control proceeds to step 346, where all of the tables and indices are stopped to allow exclusive access. Control proceeds to step 348, where a series of AMS statements similar to those in step 332 are developed. Control proceeds to step 350, where the AMS statements built in step 348 are started. Control then pro-

ceeds to step 352, where once again the checkpoints are removed from the checkpoint file and the STEP_MUST_COMPLETE flag is set. Control then proceeds to step 354, which indicates that the task is completed.

A simplified version of the table reorder sequence 400a is shown in FIG. 2. A simplified version is used here for illustrative purposes, with more detail provided below in the description of sequence 400b. In step 402 the tablespace, that is the data, is defined to be data set A and is read into the memory of the computer. There are various steps in step 404 which then logically assign new record IDs or RIDs to each of the rows of the tablespace data which have been read. In step 406 in this simplified format a table referred to as TSRID1 buffer for the clustering index which is being reordered is built. This is a table which is in indexed order and will include the new RID value for each row of data. When the buffer has been completed, it is written to disk for later use. Control proceeds to step 408, where a checkpoint is written, indicating completion of the TSRID1 buffer development operation. After this is done, the tablespace data is written out in step 410 in its new order as data set B. Control then proceeds to step 412 where a checkpoint is written to indicate completion of the data set B operation. Step 412 is a return or complete operation. The various steps of this sequence 400a are actually more detailed but are provided in this simplified version to show that the original data space is read in from the original or A file and is written out as a new B data set to allow the restarting operations as will be described below.

The restart tablespace reorg sequence 500 is shown in FIGS. 3A, 3B and 3C. The sequence 500 is utilized in case the tablespace reorganization operations do not complete once they have been started. This is not an unusual event because of the great length and complexity of the various tablespaces. Because of the millions of records which may be in a tablespace, even given the speed of the technique according to the present invention, it still a multi-hour process in many cases and thus numerous events can occur which could cause the sequences to stop prior to completion. Once operation has stopped before completion, it is desirable to be able to go back and restart the process, either by recovering and going back as quickly as possible to operational status or by restarting the process from approximately the failure point so that prior efforts do not have to be completely duplicated.

The sequence 500 commences operation at step 502 to determine if the rename sequence for each of the indices had been started. This is done by reviewing the checkpoint file. As noted above, once a particular index or table has been finished, the checkpoints are deleted from the checkpoint file so that only tables or indices which are interrupted during execution will have checkpoint entries. If the rename start has been checkpointed, control proceeds to step 504, where the rename member list is read and a pointer set to the beginning. Control then proceeds to step 506 to determine if there had been a prior restart or recover. Again it is quite possible that the restart procedure itself could be terminated abnormally, which may result in different operations upon a second or repetitive restart. If there was no prior restart or recover, control proceeds to step 508 to determine if the restart or the recover option has been selected. Generally a recover operation is one deemed to request getting the system fully operational as quickly as possible, while a restart operation is one requesting to restart the terminated operation and complete the reorganization.

If a prior restart was indicated in step 506 or a restart was indicated in step 508, control proceeds to step 510 to

determine if the A to C rename checkpoint exists. If so, control proceeds to step 512 to determine if the A to C renaming process had completed. If not, control proceeds to step 514 where the A to C renaming process is continued to completion. If the A to C rename checkpoint did not exist in step 510, the rename was complete in step 512 or after step 514, control proceeds to step 515 to determine if the B to A rename checkpoint exists. If not, control proceeds to step 531. If so, control proceeds to step 516 to check for the presence of A and B files. As will be recalled, the A files are the original files and the B files are the newly completed reorganized files. If both exist, control proceeds to step 518, where AMS statements are built to rename the particular A files to X files. Control then proceeds to step 520, which is also where control proceeds if no A files are found. In step 520 AMS statements are built to rename the particular B files which are present to A files. Control then proceeds to step 522, which is also where control proceeds if there were no B files as determined in step 516. In step 522 AMS statements are built to delete all of the X files are present. Control then proceeds to step 524 to determine if this was the last entry in the particular rename member list. If not, control proceeds to step 526 where the next entry in the rename member list is reviewed and control returns to step 516 to complete this process.

If this was the last entry in the rename member list, control proceeds to step 528 where all of the table and indices are stopped to allow exclusive access. Control then proceeds to step 530 where the AMS statements are executed and the appropriate checkpoints are removed from the checkpoint file. Control then proceeds to step 532 to determine if any partition or tablespace has been completed. If not, control proceeds to step 533 to determine if a restart or recover operation is in process. If a recover, control proceeds to step 558. If a restart, control proceeds to step 534 where the tablespace reorganization procedure must be restarted from the beginning. If the partition write was complete, control proceeds to step 536 to determine if the TSRID1 buffer had been written to disk, indicating a certain level of completion of the index and table operations. If so, the TSRID1 buffer is reloaded into memory in step 538. In step 540, a test is made to determine if there was an error loading the TSRID1 file. If so, control proceeds to step 542, which is also where control proceeds if the TSRID1 file had not been written in step 536. In step 542 a determination is made whether any indices are yet to be completed and there are further yet to do. If so, control proceeds to step 544 to determine if the STEP_MUST_COMPLETE flag is set. If so, control proceeds to step 546 as this is an error condition. If not, control proceeds to step 534 where the tablespace reorganization is restarted.

If there was no error in loading the file in step 540 or there were no indices yet to do in step 542, control proceeds to step 547 to determine if a restart or recover operation is in progress. If a restart, control proceeds to step 548, where the object checkpoints, table or index as the case may be, are reviewed and a list of objects not yet completed is developed. If a recover in step 547, control proceeds to step 549 where the list is developed only of all non-partitioned indices. Control proceeds from steps 548 and 549 to step 550, where a rename member list is rebuilt from the completed checkpoints. Control then proceeds to step 552 to determine if there is a clustering index. If not, control proceeds to step 340 in the tablespace reorganization sequence 300 to complete the operations. If there is a clustering index, control proceeds to step 326 where the clustering index operations are performed.

If in step 502 it was determined that the rename start had not been checkpointed, control proceeds to step control proceeds to step 531. In step 531 the checkpoint file is read and then control proceeds to step 532.

If in steps 506 or 508 it was determined that recover was requested or previously done, control proceeds to step 562 to determine if the A to C rename checkpoint exists. If so, control proceeds to step 564 to determine if the A to C checkpoint completion has been indicated. If not, control proceeds to step 566 where the C files are renamed to A files and control then proceeds to step 558 to delete all of the B files. If the A to C renaming has been completed, control proceeds to step 568 to determine if the STEP_MUST_COMPLETE flag is set. If so, control proceeds to step 516 as a recover can only occur by completion of the reorganization process. If the STEP_MUST_COMPLETE flag was not set in step 568, control proceeds to step 566.

If the A to C checkpoint did not exist in step 562, control proceeds to step 563 to determine if the B to A checkpoint exists. If not, control proceeds to step 531. If so, control proceeds to step 572 to determine if there are any X files. If not, control proceeds to step 574 where AMS statements are built to delete the B files. If so, control proceeds to step 574 to determine if there are any B files. If not, control proceeds to step 578 where AMS statements are built to delete the A files and to rename the X files to A files. If there are B files, control process to step 580 to determine if there are any A files. If not, control proceeds to step 582 to develop a series of AMS statements to rename all of the X files to A files and to build AMS statements to delete all of the B files. If there are A files in step 580, control proceeds to step 584 as this is an error condition. After steps 574, 578 or 582, control proceeds to step 586 to determine if this was the last entry in the rename member list. If not, control proceeds to step 588 where the next entry is obtained and control returns to step 582. If it was the last entry control proceeds to step 528.

Various detailed reorganization and reordering sequences are utilized. Those will be explained at this time. First, the non-clustering index reorganization is explained to illustrate an exemplary reorganization procedure for the index, followed by the detailed operation of the table reordering and clustering reorganization process. It is noted that no conventional sort techniques are used, and basically only sequential access is used when reading or writing files, thus greatly improving speeds, both in file access and in reordering.

Referring now to FIG. 4, a balanced tree structure such as used in a DB2 index file is shown. A root page R is at the top of the tree and contains entries directing to the two tree pages T1 and T2. Each of the tree pages T1 and T2 contain a plurality of entries which point to various leaf pages L1-L5. Each leaf page then also finally contains a series of entries which directly point to the table space pages in the table with which the index is associated. It is understood that in a DB2 file the entries in each individual leaf page L1-L5 are kept in logical order by the DB2 system itself. Similarly, the entries in each of the tree pages T1 and T2 and the root page R are also similarly automatically kept ordered in logical order within the page itself.

FIG. 5 is a sequential representation of a DB2 index file, which is a VSAM data set in either ESDS or LDS format. It is understood that when various initials are utilized, these are considered well known to those skilled in the art of IBM mainframes and DB2 and thus the full acronym may not be provided. The index file, generally referred to by the letter I, in FIG. 5 has a first page 0 which is a header page. This

header page contains particular information needed to indicate to the various packages and programs, most particularly to the operating system of the machine, the type of file and other related information. The second page of the index file I is a space map which indicates which particular subsequent pages in the file are currently active for that particular index file, these pages either being tree pages or leaf pages. The third page, page 2, is the root page R. The remaining pages in the index file I are tree, leaf, empty and unused pages in whatever order has developed through the use of the DB2 indexing process.

FIG. 6 illustrates the organization of information inside a particular root or tree page. The page starts with a header which contains various control information needed by DB2. Following this is a series of page number and key value entries, with the final entry being a single page number entry and the page concluding with an EOP or end of page indicator. For each of the series of page number and key value entries, the page number indicates the particular physical page which contains the next level down in the tree structure and the key value indicates the highest index key value contained in that particular page. The final page number in the particular root or tree page does not need a key value, as no further reference is necessary to higher key values. It is of course understood that if the index is sufficiently small, no tree pages are actually present and the root page can actually be a leaf page, but this is an unusual condition in that the database is then very small. A much more conventional structure, as indicated before, has multiple layers of tree pages between the root page and the leaf pages. If the particular structure of the index file had three levels, then the root page would indicate the appropriate number of references to specific tree pages, which tree pages would then also contain entries to another level of tree pages, which tree pages would then finally point to the leaf pages. It is also noted that for the examples in this specification show only a very limited number of entries in each page for simplicity. It is understood that in actual use a DB2 index file root or tree page will contain up to 1000 entries depending on the length of the key value, while the leaf files may contain up to 800 entries as described below, though normally a tree page has approximately 100 entries and a leaf page has approximately 70 entries.

FIGS. 7A and 7B illustrate the two various arrangements of leaf pages allowable in a DB2 index file. In the version shown in FIG. 7A, referred to as a segmented leaf page LS, the page contains a header which contains control information, followed by a subpage directory, which is a series of offsets to the particular subpages contained in the particular page and to the various high key values. Then follows a series of subpages followed by an end of page indicator. Shown below the leaf page LS is a single subpage SP which also contains a header for control information and a plurality of entries which are the actual index values. It is understood that each of the subpages has the structure as in the subpage SP. FIG. 7B shows an alternative non-segmented leaf page LN. The non-segmented leaf page LN contains a header for the necessary control information and then simply a plurality of entries which are the index entries. Therefore FIGS. 6, 7A and 7B show the structure of the various page entries present in the index file I.

FIG. 8 is an exemplary representation showing particular root and tree pages R, T1 and T2, wherein the particular physical page numbers have been illustrated. Again it is noted that this is a very simple example utilizing only 13 leaf pages, 2 tree pages and a single root page. Of course, in a typical index file there would be thousands to hundreds of

thousands to millions of these particular pages. However, the particular example is used to indicate the method of the preferred embodiment in a much more clear and concise format. In the example of FIG. 8, the root page R is page 2 of the index file I as indicated. The first page number entry, i.e. the first logical tree page, is page 9. Proceeding then to the tree page at page 9, which is tree page T1, tree page T1 contains a series of page number entries. These page number entries are the particular leaf pages in logical order. The six entries in the particular tree page T1 are the physical number of the particular page of the leaf page, while the ordering is from first to last within the particular tree page T1. Similarly, the second tree page T2, which is physical page 5 in the example of FIG. 8, contains five leaf page entries contained in logical order. It can be seen that the physical order of the entries in both the tree pages T1 and T2 and the root page R do not bear any necessary physical relationship at this time. This often develops during use of a particular database in DB2 as items are deleted, rearranged, added and so forth. Looking at the structure of FIG. 8, it can be seen that if a direct access storage device (DASD) were to try and access the various leaf pages in logical order, then the device would be skipping from page 8 to page 10 to page 3 to page 6 and so on in the index file I. All of this random access movement is necessarily slower than the various sequential movements. For example, in the case of a disk drive unit the head would have to be moved to several different locations, i.e. there would be numerous seek requests across several tracks. This is one of the slower operations of a disk drive, so performance of the DB2 system would be reduced.

FIG. 9 shows the development of the first buffer, a logical order buffer LB, utilized in the preferred embodiment. The buffer LB is preferably has a length of the number of leaf pages, in the case of FIG. 9, 11 pages. This buffer LB contains the association of the logical relationship of a particular leaf page to its physical relationship. This is developed by scanning down from the root page R through the various tree pages T1 and T2 in the following sequence. Looking at the root page R, the entries in FIG. 8 indicate tree pages at physical pages 9 and 5. Proceeding to page 9, this is tree page T1, which then has entries 8, 10, 3, 6, 11 and 4. After processing of this tree page T1 has been completed, physical page 5, which is tree page T2, is processed. Tree page T2, has in logical order the leaf pages 7, 14, 15, 12, and 13. Referencing then physical page 8, a review indicates that page 8 and thus the pages on that level are leaf pages and we have reached the effective bottom of the tree. Therefore the particular buffer LB shown in FIG. 9 has a number 8 for the physical page of the logically first leaf page in the first position. The buffer LB then proceeds to contain 10, 3, 6, 11, 4, 7, 14, 15, 12, and 13 as that is the order indicated in the tree pages T1 and T2. These leaf page values are then entered into the buffer LB of FIG. 9 in order. Therefore the buffer LB contains in each effective logical position the physical page location of the particular leaf page having that logical ranking.

The buffer LB is shown transposed or converted into a physical buffer PB in FIG. 10. The physical buffer PB is preferably first cleared so that the various tree and root entries do not receive valid data values and then the logical buffer LB is transposed. The transposition of the logical buffer LB to the physical buffer PB is such that the logical buffer LB is scanned and the logical page value is placed into the physical page number location in the physical buffer PB. For instance, physical location 8 receives the value 1 to indicate that it is the first logical ordered leaf page. This process proceeds for all the entire leaf pages. It is noted that

in FIG. 10 that there are various zeroes indicating empty entries in the physical buffer. These are the locations of the empty pages, the various system pages, the root page and the various tree pages. For example pages 2, 5 and 9 are the root and tree pages.

With this buffer PB then developed, a large buffer B is set aside in the memory space of the computer to receive the leaf page information. This large buffer set aside in the memory space has a number of pages equal to the number of leaf pages which are contained in the index file I. In the example, 15 leaf pages are shown in FIG. 11A when in actuality only 11 are necessary for the index file I of FIG. 8. With the physical buffer PB developed, the index file I is then read sequentially. Sequential operations are preferred as this the fastest operation for the DASD device. As the index file I is read sequentially, the particular physical page number of each leaf page is compared against the logical value as indicated by the physical buffer PB. If the value is 0, the not defined value, then this is an empty page, a system page, a root page or a tree page and is not to be entered into the large buffer B. In the given example of the physical buffer PB in FIG. 10, the first valid entry is physical page 3, which is also logical page 3. This particular leaf page is then written into page 3 of the large buffer B. Therefore physical page 3 has been written into what is logical page 3 of the large buffer B, as shown in FIG. 11A.

Proceeding, the next value read is physical page 4, which is logical page 6. Thus the leaf page 4 is placed into the large buffer B page 6 as shown in FIG. 11B, so that logical ordering is beginning to occur. The next page read from the index file I is page 5 and there is a zero value in the physical buffer PB, so this is a tree page. The next physical page read is page 6, which has an associated logical value of 4 and therefore as shown in FIG. 11C the contents of leaf page 6 are placed in memory space page 4. This proceeds on through the various pages as shown in the sequence FIGS. 11D-11K. By FIG. 11K the entire index file I has been read, so that all the actual leaf page information is present in the large buffer B in its proper logical order, having been read from the physical location and transferred according to the physical buffer PB into the desired logical location. Therefore the large buffer B as indicated in FIG. 11K has all of the leaf pages in proper logical order.

It is then appropriate to write the leaf pages back to a new index file, building the particular root and tree pages as they fill up. Therefore the new index file structure would be as shown in FIG. 12. Again the root page R is always page 2, while in this case the first tree page T1 is page 9. It is noted that the particular entries in tree page T1 are the various leaf pages and they are in sequential order, i.e. 3, 4, 5, 6, 7 and 8. As the 8th page was the final leaf page for the particular tree page T1, and it filled the tree page, then the tree page T1 is written in page 9. Beginning at page 10 is the next series of sequential leaf pages. After they have all be written, in the example then the tree page T2 is written to page 14, which completes the operation of writing the particular index file. It is again noted that in actual use this writing process will be significantly more complex as there will be numerous levels of trees.

Therefore it can be seen through the processes shown from FIG. 9 to FIG. 10 to FIGS. 11A-11K and then to FIG. 12 that the leaf pages have been very quickly and very simply transferred so that the index file is reorganized from its original physically scattered logical order to a sequential physical and logical order so that DASD operations are greatly enhanced for logical traverses through the index file I. Therefore operations of the DB2 system are enhanced as

desired. It is also noted that no conventional sort operation is used but rather only reads through the various tree levels to determine the leaf logical order, then a sequential reading of the leaf pages into memory followed by a writing from memory into an index file which was developed. It is noted that the two major DASD operations, the read to obtain all of the leaf pages and the write to develop the new index file, are both sequential operations. No major random operations are necessary after the development of the logical buffer LB. This greatly enhances performance, particularly as compared to a sorting operation which often utilizes many random accesses of the DASD device.

The flowcharts of FIGS. 13A, 13B, 14A and 14B illustrate sequences utilized to practice the method described above. The index reorganization sequence 100 is commenced to reorganize or reorder the particular index file. In step 102 the computer obtains the name of the particular index to be organized. In step 104 DB2 is called to determine the actual file name of the index file so that other system tools can be used to determine information about the index file. The file name is the A or C file, as appropriate. In step 105, if the INCONSISTENT STATE flag is set and the index is a non-partitioned index, the file name is changed from an A file to a C file. Having obtained the actual file name and done any necessary renaming, in step 106 the various system functions present on the computer are utilized to determine the actual DASD location of the index file and its size. This is so that simple file operations can be utilized without having to use the actual DB2 operations. Control then proceeds to step 108, where the index file is opened for random access. This is necessary to develop the tree page structure so that the logical buffer LB can be developed.

Control proceeds to step 110, where page 2, the root page, is read. In step 112 a determination is made as to whether page 2 is a leaf page. If so, this is a very small file. In the more conventional case, it is not a leaf page and control proceeds to step 114 to determine the number of pages immediately below what is now a root page. This is done by reading the number of particular page number entries in the root page. Then a first buffer is set up having that number of pages. After setting up the buffer in step 116, control proceeds to step 118 where the page numbers of the tree pages in logical order are written into this buffer. This then provides a first ordering of the tree, i.e. which branches are to be read first. Control then proceeds to step 120 so that a next level buffer can be developed. Preferably the buffer size is reserved according to the formula

$$n = p(4049/(k+3)+1)$$

where n is the number of pages at the next level, p is the number of pages at the current level and k is the length of the key field used in the particular index file. This allows an estimated buffer size that is at least as long as necessary to allow development of the ordering.

Control then proceeds to step 122 to read the first page indicated by the current level buffer written in step 118. If this particular page is a leaf page as determined in step 124, then the tree has been fully traversed and the information present in the current level buffer is the logical mapping, so that the current level buffer is the logical buffer LB. If this is not the case and it is another tree page, at least one more level must be traversed and so control proceeds to step 126 where the page numbers in logical order for the current tree page of the current level are read and placed into the next level buffer which has been developed. The current tree page is incremented so that the next actual tree page is read.

Control then proceeds to step 128 to determine if the last tree page in the current level was read. If not, control returns to step 126. By this manner the entire current tree level is traversed until all of the pages in logical order at the next lower level are present in logical order in the next level buffer being filled.

If the last tree page had been read in step 128, control returns to step 120 where the next level buffer is again developed. Control proceeds to step 122 to again read the first page indicated by the current level buffer. In this case it is assumed to be a leaf page as we have traversed to the end and control then proceeds to step 130, which is also where control proceeds from step 112 if the root page was indeed was a leaf page. In step 130 the physical buffer PB is set up such that one word is present for each page in the index file and all the values in the physical buffer PB are zeroed to allow indication of the various root and tree pages when the index file is read. Control then proceeds to step 132 (FIG. 13B) where the transposition from the logical buffer LB to the physical buffer PB is performed. This is the transposition as shown from FIG. 9 to FIG. 10. After the transposition has been completed in step 132, control proceeds to step 134 where the index file is closed as a random access file and is opened as a sequential access file to allow fast and optimal reading of the index file I. Control then proceeds to step 136 where a non-clustering index write routine 200 (FIG. 14A) is commenced. As the particular entries will have to ultimately be written to a new index file, the B file, it is desirable that the various write operations occur as concurrently as possible with the particular read operations where the large buffer is filled. In the normal case the particular index files can be quite long and generally are not horribly disorganized, so it is usually possible to begin writing the early portions of the index file structure and the early leaf pages while the order sequence 100 is completing the final pages. This allows some overlapping given the length of the particular index files and helps speed overall operations.

Control then proceeds to step 138 where the large buffer B is set aside in the memory, with one page set aside per leaf page of the index file. Control then proceeds to step 140 where a mapping table is set up with one byte available per leaf page. The values of the map table are also cleared in step 140. The map table is used as a pointer to indicate whether a particular page in the large buffer B has been filled. This is used by the write sequence 200 to determine when the next block of leaf pages can be written to the new index file. Control then proceeds to step 142 where the next page in the current index file which is being reorganized is sequentially read. Control proceeds to step 144 where the physical page number is used as a look up into the physical buffer PB to determine the logical order of this particular leaf page. Control proceeds to step 146 to determine if the page is a tree page. If so, control proceeds to step 148. If not, control proceeds to step 150 where the leaf page is written into the logical page number location in the large buffer B and a value of FF is written to the associated map table location to indicate that this page in the large buffer B has been occupied. Control then proceeds to step 148, where the sequential page number is incremented. Control then proceeds to step 152 to determine if the last page in the index file has been read. If not, control returns to step 142 and the process continues until all of the leaf pages have been read in physical order and placed in logical order in the large buffer B. If the last page was read, control proceeds to step 154 which is the end of the order sequence 100.

As noted above, the index write sequence 200 (FIG. 14A) is used to write the leaf pages which have been placed into

the large buffer B back into a new index file which is then converted to be the index file that has been reorganized. The non-clustering index write sequence 200 commences at step 202 where the names of the new or B and old or A index files are obtained. Control proceeds to step 204 to determine the actual file name of the old index file, in a manner similar to that of step 104. Control proceeds to step 206 where the actual file size and type of the old index file is obtained in a manner like that of step 106. Control proceeds to step 208 where a new or B index file of the same type and size, but having a different name, is set up. This new index file will become the reorganized index file. Control then proceeds to step 210, where this B index file is set up for sequential write operations. In step 212 the header information and space map information is written to pages 0 and 1. Control then proceeds to step 214 where a dummy root page is written to page 2 of the new index file. As the root page is not determined until one of the last steps, as it is the highest level and all the tree pages must have been developed, a dummy must be written in this case to allow the sequential operations. Control then proceeds to step 216 where a map table pointer is set to a value of 1. This is so that tracking through the map table can be developed. Control then proceeds to step 218 where the next 16 table entries in the map table are checked to determine if they are all equal to the value FF. If not, control proceeds to step 220 (FIG. 14B) to determine if all of the input pages have been processed. If not, control proceeds to step 222 to wait momentarily as now the write sequence 200 has gotten ahead of the read operation and placement in memory of the order sequence 100. Control proceeds from step 222 to step 218 so that this process continues until 16 entries are available.

If 16 entries were available in step 218, control proceeds to step 224, where the map table value is incremented by 16 and a pointer is indicated to a first buffer which is used to build leaf pages. Control then proceeds to step 226 where the 16 leaf pages are obtained from the large buffer B and provided to a DB2 rebuild operation and finally written sequentially to the new index file. It is noted that any RID values contained in the index must first be translated from old RID values to new RID values before the values are written as the RID values will change because the data table is also being reorganized. The translation is done using either the TSRID1 or TSRID3 buffers described below. Control then proceeds to step 228 to add the highest key value contained for those particular 16 leaf pages to the tree page of the next level up which is slowly being developed. Control then proceeds to step 230 to determine if the particular tree page which is being developed is full. If so, control proceeds to step 232 where the tree page is then written to the index file. If not, control proceeds to step 234 to determine if any more of the 16 input pages which are being transferred are present. If so, control proceeds to step 226 and the leaf page is rebuilt according to DB2 format and written sequentially. If there are no more pages from the 16 particular pages being written, control proceeds to step 220 to again determine if all the input pages are completed. Assuming they are not, control proceeds to step 222 and then to step 218 so that in this manner the write sequence 200 generally tracks and walks up the large buffer B until all of the leaf pages have been written.

If in step 220 all of the input pages have been processed, control proceeds to step 236 where the final leaf page is written to the index file. Control then proceeds to step 238 where this last leaf page is written to the previous prior level up tree page. Control then proceeds to step 240 to determine if the tree page was a root page. If the tree page which

17

has now been completed is not a root page, control proceeds to step 242 where the tree page is written into the new index file. Control then returns to step 238 where this final entry is then placed into the tree page at again the next level up and control then proceeds to step 240. This loop is continued until finally all of the tree pages have been completed and the final entry is ready to be made into the root page.

When the root page receives its final entry, control proceeds from step 240 to step 244 where the new index file is closed as a sequential write and is opened as a random write because now it is time to write the appropriate root page. Control then proceeds to step 246 where the developed root page is written to page 2 of the index file I. Control proceeds to step 248 where random access to the new index file is closed and then on to step 250 where the memory utilized by the large buffer, the map table and various other buffers is released. Control then proceeds to step 252 where the completion of the write operation is checkpointed. Control then proceeds to step 254 which is the end of the write sequence 200.

In a possible enhancement, it may be appropriate to scan the logical buffer LB and obtain the first few logical leaf pages in a random fashion before beginning the sequential operation of step 142 and on. One appropriate case for this operation would be when a high percentage of the first logical pages are near the physical end of the index file. If this is the case, the write sequence 200 will have little actual concurrency with the order sequence 100, as the write sequence 200 would have to wait until the end of the index file is read into the large buffer B before the first 16 entries are available for writing to the new index file. In this case, where there is a great disparity between logical and physical order for a short period, it may be more efficient to randomly read some leaf pages to allow greater actual concurrency between the order and write sequences 100 and 200 and a reduced total time.

A disorganized table is a tablespace is shown in FIG. 15A. The row entries represent the key values of the clustering index, which should in order in an organized table. Again note that only a very simple example is utilized for illustrative purposes. FIG. 15B represents the table in organized or order form, where the records or data are sequentially organized. FIGS. 16A and 16B show representations of a clustering index or CLIX buffer for the table of FIGS. 15A and 15B. FIG. 16A shows the CLIX buffer in disorganized format, wherein the keys are sequential but the RIDs are jumbled. FIG. 16B shows the CLIX buffer if the table were properly organized. The CLIX buffer is one buffer utilized during the table reorder and clustering index reorganization process.

The detailed tablespace reorder sequence 400b (FIG. 17A) commences at step 1402 where a determination is made as to the type of tablespace. If it is a simple tablespace, a segmented tablespace with a single table or a partitioned tablespace, control proceeds to step 1404 where the clustering index of the particular table being reordered is determined. Control then proceeds to step 1406 where steps 104 to 134 and 138 to 152 of the non-clustering index reorganization sequence 100 are performed to develop the required order of the keys in the clustering index. This thus provides the CLIX buffer which includes the key numbers in sequence and the associated old RID numbers. Control then proceeds to step 1408 where a TSRID2 buffer is set up as two columns, the first for the old RID number and the second for the new RID number, the number of rows equalling the number of rows in the table. While this information could be obtained from the CLIX buffer, the use of the TSRID2 buffer

18

is preferred for speed reasons. Each entry in the CLIX buffer may actually be quite large, as the key may have a maximum length of 256 bytes, while the TSRID2 buffer uses only 4 bytes per column entry or 8 bytes per row. This smaller size greatly improves processing speeds. Control then proceeds to step 1410 where the CLIX buffer is read sequentially to obtain the old RID value, which is written to the old RID column of the TSRID2 buffer, thus properly sequencing the TSRID2 buffer. Concurrently with this reading and writing operation the RID values are monitored for the largest relative row number in any particular page and this row number is assigned to a variable referred to as MAXMAPID.

Control then proceeds to step 1412 where a TSRID1 buffer is constructed having a number of slots in the buffer equal to the number of pages of the particular table times the MAXMAPID variable value. Control then proceeds to step 1414, where a table buffer is set up in memory having an area for the data. Control then proceeds to step 1416 where the table data is sequentially read into the table buffer in physical order, skipping any empty pages and saving the data according to a compressed format in the reserved memory area.

As the total size of the table data can be large, space saving techniques are preferably used when the data is stored. The preferred technique includes grouping pages into segments of 16 pages each, with each segment starting on a 32 byte boundary. The relative number of this 32 byte boundary is saved in a memory area referred to as the segment reference table. The segment reference table thus contains one word per 16 pages of table data and the value is the relative 32 byte cell number where a particular segment begins. Each segment area in turn begins with 16 one word offsets to the beginning of the data for that page relative to the beginning of the segment. Therefore, to find the beginning of the data for a particular page, the page number is divided by 16 and this value multiplied by 4 to provide the offset into the segment reference table. The value at this location is the cell number to the particular segment data which, when multiplied by 32, is the offset into the table data where the segment begins. Any remainder from dividing the page number by 16 is the relative page number of the segment and when multiplied by 4 is the offset into the segment header which is used to obtain the offset in the segment which indicates where the actual page data begins. Thus a particular page can be accessed without searching using only a few computations.

The data for the particular page is saved in a format which also allows access to an individual row without searching. As DB2 keeps MAPIDs at the end of each page, the MAPIDs being two byte offsets computed from the beginning of the page for each of the table rows. It is noted that the MAPIDs are numbered sequentially from the end of the page forward. The MAPIDs can then be used for computation purposes. Preferably the MAPIDs are placed at the beginning of each page of data, after a prefix which indicates the length of the MAPID section. To access a specific row of data, the process determines the MAPID by determining the page address plus the prefix value minus twice the row number. The MAPID value, plus the page address plus the page length minus 20 produces the address to the actual row data. Therefore a particular row within a page can also be determined without searching.

Additionally in step 1416, the length of each row is copied in order from the table buffer to the TSRID1 buffer unless the RID value indicates an overflow row, a row which has been replaced and cannot fit in the originally allotted space, in which case an OVERFLOW flag is written. This will

allow later reference to determine the actual RID value for the overflow row. Alternatively, if the row is a deleted row, a DELETED flag is written and the row is not written to the table buffer. The TSRID1 buffer slot location is determined by multiplying the page number, as indicated in the RID, times the MAXMAPID variable and adding the row number. The CLIX, TSRID2 and TSRID1 buffers are shown in a simple example in FIG. 18.

Control proceeds from step 1418 to step 1420 (FIG. 17B) where a pointer is set to the beginning of the TSRID2 buffer. Control then proceeds to read the TSRID2 buffer to contain the old RID value and then to calculate the slot in the particular TSRID1 buffer as noted above to obtain the record length for that row. Control proceeds to step 1424 to determine if the OVERFLOW flag was set. If so, control proceeds to step 1426 where the overflow RID value is used to retrieve the actual row length from the table row data, entry calculated as noted above. The actual overflow RID value is saved in the TSRID2 buffer slot as the old RID value. After step 1426 or if there is not an overflow RID as indicated in step 1424, control proceeds to step 1428 where the row length value is used to assign a new RID value for each row. The row length is utilized because a particular page has only a given size and once the given page has been filled up, a new page must be used. Thus the row lengths are added for each page until an overflow occurs, at which time a new page is allocated. As this task proceeds, the relative row number is determined by the entry number in the page and the page number is tracked in an incrementing manner. However, if the DELETED flag is set, then the TSRID2 buffer also receives the DELETED flag and a new RID value is not determined.

After the new RID value has been obtained in step 1428, it is saved as the new RID value in the TSRID1 and TSRID2 buffer slots for that row. Note that this replaces the page length value in the TSRID1 buffer. This process is shown in intermediate and final state in FIGS. 19A and 19B. Note that new RID values are being added to the TSRID2 buffer and new RID values are replacing row lengths in the TSRID1 buffer. Control then proceeds to step 1430, where header page and space map pages are constructed in a separate memory area when there is sufficient data to develop the particular page. The header page is developed immediately upon the first operation and the space map pages are developed as pages are filled up. Control then proceeds to step 1432 where the pointer to the TSRID2 buffer is incremented. Control proceeds to step 1434 to determine if the RID calculation operation using the TSRID2 buffer has been completed. If not, control returns to step 1422 and the sequence is performed until the entire TSRID2 buffer has been traversed and all of the new RID values have been determined.

When all the new RID values have been determined, control proceeds to step 1436 where the TSRID1 buffer is written to a file and completion of the TSRID1 buffer is checkpointed. It is noted that the TSRID1 file thus contains the new RIDs in the sequential order of the keys. Control then proceeds to step 1438 where steps 202 to 252 are performed for the clustering index, noting of course that the old RID values are translated to the new RID values by reference to the new RID column in TSRID2 buffer for each particular key. Control then proceeds to step 1440, where a table B file is opened for sequential operation. In step 1442 the header page and the first space map page are written to the table B. Proceeding then to step 1444, the row data is written in order for the space map page, using the TSRID2 buffer in sequential order as the source of the old RID value,

with the old RID value being utilized to calculate the entry into the memory area for the particular row of data. The MAPIDs are developed as the write operation progresses. Any gaps in the new RID numbers as noted in the TSRID2 buffer are free pages with all data in the row being 0 this process continues until the next space map page is to be written. Control then proceeds to step 1445 to determine if all the pages have been written. If not, control proceeds to step 1446 where the next space map page is retrieved from memory and written to the table B file. Control then proceeds to step 1444 and the data for that page is written. This process repeats until all the data has been written. Control proceeds from step 1445 to step 1448 when all the pages have been written. In step 1448 sequential access to the table B file is closed and the completion of the table B write operation is checkpointed.

If it was determined in step 1402 that this is a segmented tablespace with multiple tables, control proceeds to step 1450 where a table buffer is set up having an area to receive the actual data. Additionally, an area is set up for a TSRID3 buffer, with that buffer having one column with slots for each row and an additional two slots for each page. Control then proceeds to step 1452, where the table data is sequentially read into the table buffer in physical order, skipping any empty columns and saving the data according to the compressed format discussed above. At the same time, the length of the particular row or the overflow RID value is placed in the TSRID3 buffer slot. The row count for each page is placed in the page slot and a cell range list for each separate table is maintained. A cell range list keeps track of the pages in each segment. An exemplary TSRID3 buffer is shown in FIG. 20A. Control then proceeds to step 1454, where a pointer is set to indicate the first segment or table. Control then proceeds to step 1456 to determine if this is an indexed table. If not, control proceeds to step 1458. In step 1458 by referencing the cell range list and the page length in the TSRID3 buffer, new RID values are assigned as noted above, which then replace the length value in the TSRID3 buffer, building the space map pages as appropriate and saving those in memory. This is illustrated in FIG. 20B where an intermediate state of reorganizing the table of FIG. 20A is shown. Control proceeds to step 1460 where the table is saved in the FIFO queue. Control then proceeds to step 1462, where the pointer value is incremented. Control proceeds to step 1464 to determine if the last table was completed. If not, control returns to step 1456.

If in step 1456 it was determined that this was an indexed table, control proceeds to step 1466 where steps 104 to 134 and 138 to 152 are executed to develop the required order of the keys in a CLIX buffer. Control proceeds to step 1468, where a TSRID2 buffer is set up, having columns for the old RID and the new RID values for each row. Control proceeds to step 1470, where the old RID values are read sequentially from the CLIX buffer and written to the old RID column of the TSRID2 buffer. Control proceeds to step 1472, where by proceeding sequentially through the TSRID2 buffer and obtaining the page length from the TSRID3 buffer using the old RID value to indicate position in the buffers, new RID values are assigned and placed in the TSRID2 and TSRID3 buffers. This process is shown in sequence from FIG. 21A, where all of the values have been placed in the buffers; to FIG. 21B, where an intermediate state is shown; and to FIG. 21C, the final state of the buffers. Additionally, the space map pages are built as appropriate and saved in memory. Control proceeds to step 1474, where the table is saved in the FIFO queue. Control then proceeds to step 1476, where steps 202 to 252 are performed for the clustering index,

noting that the old RID values are translated to the new RID values by using the TSRID2 buffer. Control then proceeds to step 1462.

If at steps 1464 the last table was completed, control proceeds to step 1482 where a table B file is opened for sequential operation and to step 1484, where a pointer is set to the first table. Control proceeds to step 1486 to determine if this was an indexed table. If not, control proceeds to step 1488, where the header page and the first space map page are written to the table B file. Control then proceeds to step 1490, where the row data is written in order for the first space map page using the TSRID3 buffer in sequential order and filing in the MAPIDs as progressing as discussed above. In this case the old RID values are based on the position in the TSRID3 buffer and are used with the page table list to obtain particular row data. Any gaps in new RID numbers are filled by free pages with all zeros until the next space map page is to be written. Control then proceeds to step 1491 to determine if all the pages have been written. If not, control proceeds to step 1492, where the next space map page is written to the table B file. Step 1490 is then re-executed and this continues until all pages have been written. When all the pages have been written, control proceeds from step 1491 to step 1504, where the pointer value is incremented. Control proceeds to step 1506 to determine if the last table is completed. If not, control returns to step 1486.

If in step 1486 it was determined that it was an indexed table, control proceeds to step 1498, where the header page and the first space map page are written to the table B file. Control then proceeds to step 1500, where the row data is written in order for the first space map page using the TSRID2 buffer in sequential order as the source of the old RID value. The old RID value is used to calculate into the memory area to determine the length of the row data for determining new RID values. Any MAPIDs are developed as the process is progressing. Any gaps in the new RID numbers are filled in by free pages with all zeros, until the next space map page is to be written. Control then proceeds to step 1501 to determine if all the pages have been written. If not, control proceeds to step 1502, where the next space map page is written and then to step 1500 to repeat the page data writing operation. When all the data pages have been written, control proceeds from step 1501 to step 1504. If not, control returns to step 1486. If step 1506 determines that the last table has been completed, control proceeds to step 1508 where sequential access to the table B file is closed and the completion of the table B write operation is checkpointed. Control then proceeds to step 1510, which is a completion or end of the sequence.

As previously stated, all of the operations to the table data to and from DASD are performed sequentially, thus greatly improving DASD performance. Further, no conventional sorting is done, which is a very cumbersome and slow process. Thus a second means of speed improvement is provided. Similar statements can be made for the operations with the clustering index. Various tests have shown a performance improvement of three to six times over conventional tablespace reorganization procedures.

The above-description has made only minor reference to the actual threads and multi-tasking that can be utilized. This was done for purposes of simplifying the explanation. As conventionally done in mainframe operation, many of the illustrated operations may actually be executing concurrently or started for operation, which then proceeds according to the operating system scheduling process. This concurrency may require certain tests in certain sequences to prevent overrunning, but this is conventional and can readily be provided by one skilled in the art.

Further, the above description has focused on DB2 tablespaces, but similar techniques can be used for databases developed in other formats, particularly tree-structured formats.

The foregoing disclosure and description of the invention are illustrative and explanatory thereof, and various changes in the method of operation may be made without departing from the spirit of the invention.

We claim:

1. A method for reorganizing a data table on a computer storage system, the data table organized into pages, each page storing a number of data rows based on the size of the data row, wherein row data location includes page number and relative row number and not having a related index, the method comprising the computer executed steps of:

sequentially reading the data rows into a data buffer having an area for receiving the data and copying the length of each data row to a length buffer having an entry for each data row, said length stored in said length buffer based on the original location of the data row;

determining a new location for each data row by sequentially obtaining the row length of each data row from said length buffer and using said row length to develop said new location, including using said row length to determine entry of each data row into a page and the relative row number in that page;

saving said new location in a position corresponding to the data row original location; and

sequentially storing said data in a reorganized file by sequentially reading said data row from said data area and storing said read data row in said reorganized file.

2. The method of claim 1, wherein said step of saving said new location further saves said new location in said length buffer, replacing said row length.

3. The method of claim 1, wherein the data table further includes space map pages for identifying space in each of said pages, wherein said step of determining a new location further includes determining space map information for each new page and saving said space map information and wherein said step of storing said data includes retrieving said space map pages and storing said space map page for a given page in said reorganized file prior to storing said read row data for said given page.

4. A method for reorganizing a data table on a computer storage system, the data table being comprised of a plurality of data rows and having a related index, the method comprising the computer executed steps of:

reading the index to determine the logical order of the data and the original location of the data and storing the ordering information and original location in logical sequence in an index buffer having entries for each data row;

sequentially reading the data rows into a data buffer having an area for receiving the data and copying the length of each data row to a length buffer having entries for each data row, said length stored in said length buffer based on the original location of the data row;

determining a new location for each data row by sequentially obtaining the row length of each data row from said length buffer using said original location and using said row length to develop said new location;

saving said new location in a position corresponding to the data row original location; and

sequentially sorting said data in a reorganized file by determining the original location of each data row in logical sequence, reading said data row from said data

area using said original location and storing said read data row in said reorganized file.

5. The method of claim 4, wherein said step of reading the data rows saves said data rows in said data buffer in a compressed format allowing determination of a given data row location using only calculations and table lookups and without using searching techniques.

6. The method of claim 4, wherein said index has a tree structure with tree and leaf pages, the tree pages for indicating the logical order of tree pages and leaf pages and wherein said step of reading the index includes the steps of:

reading the tree pages to determine the logical order of the leaf pages;

preparing a buffer correlating the leaf page physical position with its logical position for each leaf page;

preparing a large buffer for receiving a copy of all the leaf pages in the index;

sequentially reading the leaf pages from the index; and

for each sequentially read leaf page, determining its logical position by reference to said physical to logical buffer and placing the read leaf page in said large buffer and placing the read leaf page in said large buffer at a location corresponding to its logical location, wherein said large buffer is said index buffer.

7. The method of claim 4, further comprising the steps of: sequentially copying the original location of the data rows into a location buffer from said index buffer, said location buffer further including a new location position associated with each original location, and

wherein said step of saving said new location saves said new location in said location buffer, whereby said original location and said new location are associated and wherein said step of storing said data determines said original location by referencing said location buffer in sequential order.

8. The method of claim 7, wherein said step of saving said new location further saves said new location in said length buffer, replacing said row length.

9. The method of claim 4, wherein said data table is further organized into pages, each page storing a number of data rows based on the size of the data row, wherein row data location includes page number and relative row number and wherein said step of determining a new location further includes using said row length to determine entry of each data row into a page and the relative row number in that page.

10. The method of claim 9, wherein the data table further includes space map pages for identifying space in each of said pages, wherein said step of determining a new location further includes determining space map information for each new page and saving said space map information and wherein said step of storing said data includes retrieving said space map pages and storing said space map page for a given page in said reorganized file prior to storing said read row data for said given page.

11. The methods of claim 9, further comprising the step of:

determining the largest relative row number in any page and the total number of pages, and wherein said length buffer has a number of entries equalling the largest relative row number times the total number of pages.

12. The method of claim 11, wherein said step of copying the length of the data row places the length in the entry corresponding to the page number times the largest relative row number plus the relative row number for the data row and wherein said step of determining a new location obtains

the length by multiplying the page number by the largest relative row number and adding the relative row number.

13. A method for reorganizing a data table and a related index on a computer storage system, the data table being comprised of a plurality of data rows, the method comprising the computer executed steps of:

reading the index to determine the logical order of the data and the original location of the data and storing the ordering information and original location in logical sequence in an index buffer having entries for each data row;

sequentially reading the data rows into a data buffer having an area for receiving the data and copying the length of each data row to a length buffer having entries for each data row, said length stored in said length buffer based on the original location of the data row;

determining a new location for each data row by sequentially obtaining the row length of each data row from said length buffer using said original location and using said row length to develop said new location;

saving said new location in a position corresponding to the data row original location;

sequentially storing the entries in said index buffer in a reorganized index file using the new locations;

sequentially storing said data in a reorganized file by determining the original location of each data row in logical sequence, reading said data row from said data area using said original location and storing said read data row in said reorganized file.

14. The method of claim 13, wherein said step of reading the data rows saves said data rows in said data buffer in a compressed format allowing determination of a given data row location using only calculations and table lookups and without using searching techniques.

15. The method of claim 13, further comprising the steps of:

sequentially copying the original location of the data rows into a location buffer from said index buffer, said location buffer further including a new location position associated with each original location, and

wherein said step of saving said new location saves said new location in said location buffer, whereby said original location and said new location are associated, wherein said step of storing the entries in said index buffer replaces said original locations with new locations and wherein said step of storing said data determines said original location by referencing said location buffer in sequential order.

16. The method of claim 15, wherein said step of saving said new location further saves said new location in said length buffer, replacing said row length.

17. The method of claim 13, wherein said index has a tree structure with tree and leaf pages, the tree pages for indicating the logical order of tree pages and leaf pages and wherein said step of reading the index includes the steps of:

reading the tree pages to determine the logical order of the leaf pages;

preparing a buffer correlating the leaf page physical position with its logical position for each leaf page;

preparing a large buffer for receiving a copy of all the leaf pages in the index;

sequentially reading the leaf pages from the index; and

for each sequentially read leaf page, determining its logical position by reference to said physical to logical buffer and placing the read leaf page in said large buffer

25

and placing the read leaf page in said large buffer at a location corresponding to its logical location, wherein said large buffer is said index buffer.

18. The method of claim 17, wherein said step of reading the index further includes the step of:

preparing a buffer correlating the leaf page logical position to its physical position for each leaf page; and wherein said step of preparing said physical to logical buffer includes transposing said logical to physical buffer.

19. The method of claim 17, wherein said step of storing the entries in said reorganized index includes the step of:

developing tree pages as the leaf pages are stored and storing said tree pages to said reorganized index as filled in sequence with the leaf pages.

20. The method of claim 13, wherein said data table is further organized into pages, each page storing a number of data rows based on the size of the data row, wherein row data location includes page number and relative row number and wherein said step of determining a new location further includes using said row length to determine entry of each data row into a page and the relative row number in that page.

26

21. The method of claim 20, wherein the data table further includes space map pages for identifying space in each of said pages, wherein said step of determining a new location further includes determining space map information for each new page and saving said space map information and wherein said step of storing said data includes retrieving said space map pages and storing said space map page for a given page in said reorganized file prior to storing said read row data for said given page.

22. The methods of claim 20, further comprising the step of:

determining the largest relative row number in any page and the total number of pages, and wherein said length buffer has a number of entries equalling the largest relative row number times the total number of pages.

23. The method of claim 22, wherein said step of copying the length of the data row places the length in the entry corresponding to the page number times the largest relative row number plus the relative row number for the data row and wherein said step of determining a new location obtains the length by multiplying the page number by the largest relative row number and adding the relative row number.

* * * * *

Appendix D

Evidence Appendix

Other than the references attached to the Appeal Brief as Appendices B and C, no evidence was submitted pursuant to 37 C.F.R. §§ 1.130, 1.131, or 1.132, and no other evidence was entered by the Examiner and relied upon by Appellant in the Appeal.

Appendix E

Related Proceedings Appendix

As stated on page 3 of this Appeal Brief, to the knowledge of Appellant's counsel, there are no known appeals, interferences, or judicial proceedings that will directly affect or be directly affected by or have a bearing on the Board's decision regarding this Appeal.